DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Design and implementation of a framework extension for automated testing of a CNC's real time fieldbus.

Relatório submetido à Universidade Federal de Santa Catarina

como requisito para a aprovação na disciplina

DAS 5511: Projeto de Fim de Curso

Arthur Sady Cordeiro Rossetti

Florianópolis, agosto de 2016

Design and implementation of a framework extension for automated system testing of a CNC's real time fieldbus

Arthur Sady Cordeiro Rossetti

Esta monografia foi julgada no contexto da disciplina
DAS5511: Projeto de Fim de Curso
e aprovada na sua forma final pelo
Curso de Engenharia de Controle e Automação

Prof. *Marcelo Ricardo Stemmer*

_____
Assinatura do Orientador

Banca Examinadora:

Pedro Reboredo, Dr-Ing.
Orientador na Empresa

Prof. Marcelo Ricardo Stemmer
Orientador no Curso

Prof. _____
Avaliador

Aluno:_____

Aluno:_____
Debatedores

Acknowledgements:

I would like to thank all the people that contributed for me to be able to reach this point in my life. I thank all the teachers and professors I had since elementary school until university, who taught me many things. I thank the friends that stood with me in the middle of the way, especially those of my class from 2010.1 without whom I believe it would take at least twice the time to be here today. And also the friends I made and lived with in Sheffield. I thank my co-workers and supervisors in all the internships I made, who helped me grow. Finally, I thank my family for teaching me how to live, my cousins, grandparents, uncles and aunts, my brother and sister and especially my mother and father, who always believed I could do anything and gave me the opportunity to try myself. This work would not be possible without all of them.

Resumo

O trabalho desenvolvido neste documento foi realizado na empresa Bosch Rexroth AG. Esta empresa, dentre muitos de seus produtos, também se especializa na criação de drives de controle e dispositivos de atuação para máquinas. Este projeto tratou especificamente com a criação de testes automáticos de conformidade para um dispositivo de comunicação de tempo real utilizado em um controlador de sistemas CNC conhecido como MTX. Os sistemas de comando numérico computadorizado que utilizam os controladores MTX são escaláveis podendo conter até 250 drives para eixos com controle de posição linear e 64 drives para eixos com controle de velocidade rotacional.

O dispositivo de comunicação em questão, utilizado por estes sistemas, é um fieldbus de tempo real baseado em ethernet desenvolvido pela companhia conhecido como “Sercos III”. Dentro dos números de dispositivos mencionados, este dispositivo consegue garantir um tempo de sincronização entre os dispositivos de até 1µs e um tempo total de ciclo entre 500µs e 8ms.

Os controladores CNC da companhia têm sido utilizados por diversas empresas em ramos diferentes, como por exemplo, usinagem, corte a laser, corte com jato d’agua, manufatura entre inúmeros outros setores como pode ser visto em [1]. Em várias destas áreas a capacidade de sincronizar as operações é crucial para garantir uma boa qualidade de produção, tornando a presença de um dispositivo Sercos indispensável.

Estas máquinas, que utilizam o mesmo sistema de controle deste projeto, antes de chegarem aos clientes e mesmo durante as fases de desenvolvimento interno da empresa, passam por diversos tipos de teste, dentre eles, os testes de sistema. Estes precisam ser executados não apenas quando uma máquina é configurada pela primeira vez, mas toda vez que acontece uma atualização significativa de software ou firmware visando garantir que nenhuma funcionalidade tenha sido comprometida. Este processo de aplicar todos os testes novamente a cada atualização é chamado de *“Regression Testing”.* A utilização de testes automáticos neste contexto visa, portanto, melhorar o desempenho das verificações realizadas nos sistemas além de garantir aumento na qualidade dos produtos.

Além de estes testes serem repetidos com alguma frequência, é interessante ressaltar que a empresa intenciona aplicar a metodologia de desenvolvimento direcionada por testes (“Test Driven Metodology”), que primeiro estabelece metas a serem alcançadas

nos testes, e posteriormente desenvolve sistemas capazes de suprir estes objetivos. Existem argumentos de que esta metodologia não é a melhor tendo em vista que apenas se deseja alcançar o patamar estipulado pelos testes e não a solução ótima do problema. Entretanto até o momento acredita-se que esta metodologia traga um custo-benefício muito interessante para o desenvolvimento de novas tecnologias da empresa. Tendo isso em vista, a capacidade de criação de novos testes torna-se, também, muito importante.

Dentro deste universo, o projeto visou expandir um framework para testes, já utilizado em outros sistemas da companhia, desenvolvido em C++ utilizando-se o ambiente de programação Visual Studio Pro da Microsoft. Além disso, foi necessária a criação de uma estrutura auxiliar que permitisse aprimorar a comunicação entre o framework e o sistema CNC, possibilitando aumentar a gama de informações e comandos que podia ser transferida entre eles. Possibilitando assim a automatização de testes já existentes, bem como facilitar sua utilização ou aperfeiçoamento no futuro e simplificar a criação de testes completamente novos. No fim, conseguiu-se atingir esses objetivos, entretanto isto não significa que o projeto na empresa tenha chegado ao fim, a equipe continuará aperfeiçoando os resultados assim como as estruturas criadas.

Por fim, durante o projeto, foi decidido comparar as vantagens da criação destes testes automáticos em duas linguagens distintas, C++ e Lua, sendo estas linguagens compilada e interpretada, respectivamente. Este ponto era de interesse para o projeto, pois os profissionais responsáveis por testes, muitas vezes não são programadores experientes. Desta forma, havia interesse em avaliar ambas as linguagens, tendo em vista que na linguagem interpretada escolhida, o tratamento de variáveis é realizado em mais alto nível e não existe a necessidade de tratar com ponteiros, o que se acreditou configurar uma característica facilitadora. Este interesse também surgiu devido à implementação de um plugin em estado de protótipo que permite que o framework execute testes em Lua. Durante o projeto descobriu-se também que o framework apenas funciona com a versão Visual Studio Pro da Microsoft, o que necessitaria uma licença pra cada usuário, enquanto que, uma vez que o framework esteja compilado, os testes em Lua podem continuar a ser alterados sem a necessidade de nenhuma IDE. Essa comparação foi realizada baseando-se na opinião do desenvolvedor deste projeto e as conclusões são apresentadas posteriormente neste documento.

Abstract

The Project developed in this document was performed at the company Bosch Rexroth AG. Amidst its wide range of products, they also specialize in the development of control drives and actuators for machines. This project was specifically related to the development of automated system tests for a real time communication device used by a CNC controller known as MTX. The computer numeric control systems that use the MTX controllers can be scaled to have up to 250 drives for axis linear interpolation and 64 drives for spindle velocity control.

The communication device mentioned is a real time fieldbus based on Ethernet, also developed by the company, known as “Sercos III”. If the number of devices respect the limits before mentioned, Sercos can ensure a synchronization time between the devices as small as 1µs with a cycle time for the whole structure that can range from 8ms to 500µs.

The CNC machines produced with the company’s equipment are being used in different fields like, milling, grinding, hydro jet, laser cutting and so on, more information can be found at [1]. In many of these fields, the capability to synchronize the operations between drives is crucial, and also need to be performed very fast in order to guarantee a good production quality and output, which makes the presence of the Sercos device imperative.

These machines, which use the same control system as the one in this project, before reaching the clients and even during the internal development in the company, are subjected to a wide variety of tests in which are included the system tests. These tests, specifically, must be performed not only when the machine is configured for the first time, but every time when there is a significant software or firmware update, aiming to ensure that no functionality has been compromised. This policy for repeating the tests for every new version of the system configuration is known as regression testing. The usage of automated testing in this universe aims to improve the quality checks performed in the systems, and guarantee a better quality for the products in the future.

Aside from the fact that these tests are repeated with some frequency, it is also important to point out that the company is targeting to work with “Test Driven Development”, in which first are defined tests and goals to be cleared by the tested systems, and afterwards the systems are developed aiming to clear those tests, even though the methodology can’t

be applied to every case. Some argue that this is not the best approach because it does not aim for the optimal solution and instead only to clear the set point established in the tests, but for the company the methodology is thought to present an interesting trade-off between quality and cost since the optimal solution is often more expensive. Taking this fact into account the ability to create new test cases becomes also really important in the project.

Considering this scope, the project tried to expand an existing framework for tests already used by the company with other systems. The framework was developed in C++ using the Visual Studio Pro from Microsoft. Moreover it was also needed to create an auxiliary structure to enhance the communication between the framework and the MTX system, increasing the applications that could be used by the automatic tests, enabling the automation of existing tests as well as simplifying its usage, its enhancing and also the creation of new tests. In the end, this objectives were met, considering what could be done during six months, but this does not mean the project in the company was finished, the team will continue to perfect the results.

Lastly, during the project it was also decided to compare the advantages or disadvantages between two programing languages, C++ and Lua, in the creation of automated tests, taking into account that one of them is compiled and the other interpreted. This topic was of interest because the users responsible for the tests may not be experienced programmers, and considering that in the interpreted language handling variables is done in a higher level, using Lua was considered a possible opportunity to make it less complex.

Another reason why it was interesting to test this language was because there was a plugin developed in another project of the company related to the framework, which is in a prototype state and enables the usage of Lua scripts to create test cases. During this project it was also discovered that, since the framework uses some proprietary libraries, it can only be compiled using Visual Studio Pro, the Express version of the software is not enough, which makes the usage of this specific IDE, mandatory, at least for now. On the other hand, once the framework is compiled, the Lua tests can still be modified without the need for any development environment at all. The comparison between both languages was made taking into account the evaluation done this project and is presented further in this document.

Summary

Simbology

A seguir:

CPL	Customer Programming Language
PLC	Programmable Logic Computer
IWE	Indraworks Engineering
IWO	Indraworks Operation
Sercos	Serial Real-Time Communication System
MDT	Master Data Telegram
AT	Acknowledge Telegram

Chapter 1: Introduction.

This document has the purpose of explaining the development of the final course project realized by the student Arthur Sady Cordeiro Rossetti from the Federal University of Santa Catarina, executed during an internship at Bosch Rexroth AG.

1.1: Scope and Objectives.

The project is an attempt to simplify the creation and operation of automated system tests, realized on a real-time fieldbus as well as to implement as many test cases as possible. For this project, the communication device, named “Sercos”, is used in association with a CNC system controller developed by the company known as “MTX”. This specific controller is composed by a PLC and a numeric controller, as well as a Sercos device, each with a wide range of operations available, enhancing flexibility and making the MTX the perfect equipment for numerous applications.

The communication between the MTX’s components and all drives for the actuators is handled by the before mentioned fieldbus, which is an equipment composed by an FPGA and a communication stack, also developed by Bosch Rexroth. Now in its third release version, “Sercos III”, the device is a real time field bus based on Ethernet. It can be used for a variety of applications, and is employed by many products from the company aside from the MTX to perform the communication or synchronization between devices. Since it uses a well-known protocol (IEC 61491), it is also supported by many other companies’ equipment making it even more versatile. Provided that the number of devices is kept below the recommended values, the synchronization interval can be as small as 0.1µs between them. In Figure 1 a simplified diagram is displayed to contextualize the interaction between the devices.

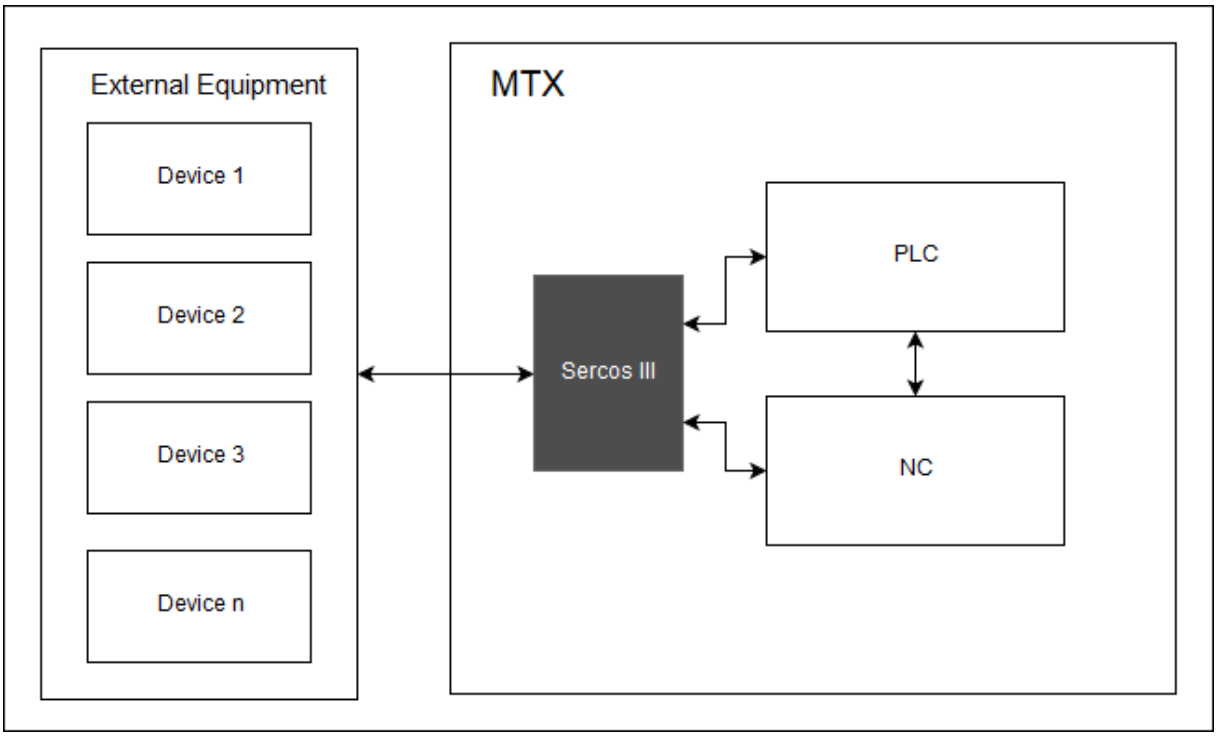

Figure 1 - MTX and Sercos Integration

Sercos is a core device in the company's portfolio because of its real time features. Taking into account its importance, it is imperative that the device's software, firmware and other specific attributes are repeatedly checked before delivery to clients and also during internal tests in the developing process. Some of the tools used for that are the system tests. They are used to guarantee that the device performs its functions as expected, meeting the necessary specifications. They can be used after configuring the device or after an update in either software or firmware to ensure the functionalities of the system weren't affected.

Historically, these system tests were of interest to the company not only for the MTX, but also regarding other systems that employ Sercos. In the past, they were done manually, usually using the company's software "Indraworks Engineering". These activities were monotonous and more often than not, simple, but time consuming. With this in mind and aiming to free workers to perform more complex tasks and better employ their time, as well as standardize the results for the tests, it was decided to automate them. This automation process was already started for other company systems, and with this project, the aim was to start it for the MTX.

1.2: Initial Situation and Plan

The company already possessed a framework used for automated tests known as “TestBuddy”, which operated with other systems. To achieve the project’s objectives, the idea was to extend this framework to work with the MTX. This would be possible because a communication interface between the framework and MTX, called NCS, was already developed through the usage of C++ libraries. It could enable TestBuddy to execute commands with the system, however, the functions provided could not deliver all the necessary functionalities desired for the automatic tests. Moreover, the libraries are heterogeneously developed and usually have poor documentation, or are developed for internal usage only, sometimes having no documentation at all.

The framework was developed using Visual Studio Pro in C++; it works based on the development of plugins, which create Dynamic Link Libraries (DLLs). These Libraries are interpreted by an executable file, which runs the TestBuddy. This way, in order to develop new applications or new modules, it is only necessary to develop a new Plugin and insert its DLL in the same directory as the executable file. This project worked over a template plugin named PluginNck, adding all the necessary libraries and functions of the project in this plugin.

Using the approach of extending the framework capabilities through the creation of new plugins, in another project, a plugin was created to allow the TestBuddy to run scripts from an interpreted programing language called Lua. This language, which was created by the Pontifical Catholic University of Rio de Janeiro in Brazil, has a diversity of applications and more specifically, is really simple and strong when working with data structures. In Figure 2 a diagram is displayed to materialize how exactly the TestBuddy is structured.

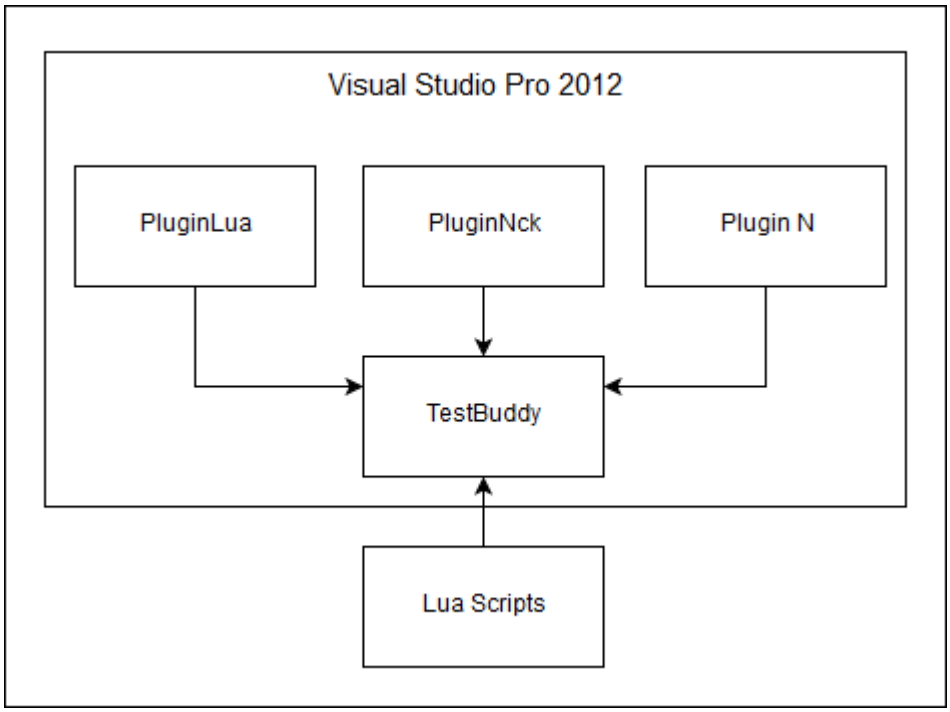

Figure 2 - TestBuddy Framework Structure

Considering the situation of the available tools mentioned for the TestBuddy, namely the NCS libraries and also the Lua plugin, two main plans were devised. First, it was considered necessary to create an auxiliary structure that would enhance the communications already possible between the framework and the MTX system. For simplicity, it was decided to develop a service structure in which the framework would be the client and the PLC and NC from the MTX would be the servers, making it possible to develop services that would complement the NCS functionalities.

This structure would use a special memory space in the MTX, which can be configured to employ data structures as the ones supported by programing languages like integers, Booleans, chars, and other simple variables as well as complex structures like arrays, structs and so on. This memory space, named as “System Data”, would be used to simulate communication channels, enabling the service architecture to exchange data with the framework.

The idea to employ this specific method which uses memory to act as middle ground between the systems was originated by the fact that this memory space is

accessible by the three of them independently. This way it would become simpler to build the desired architecture.

The service structure would be divided in two pieces for the servers; one would be implemented in the PLC and the other in the NC. To make this structure work reliably on the PLC side, it was decided to use a state machine to handle service requests. This approach was thought to be the best for two main reasons, first because it is easier to use the PLC loop structure if the operations are defined in states, and second because this way it would be less complex to keep adding new functionalities to the program without changing the whole structure.

The NC structure to process the service calls had to be simpler, due to limitations on the language it supports called CPL. The services would all be treated in a single case structure contained in a single NC program but this would not represent a big problem to the project because the functionalities needed from the NC are not as complex as the ones required from the PLC and also because the NC has direct access to the “System Data”. Simplifying a lot the needs related to variable access compared to the PLC which has to use specific function blocks to fetch information.

Then, the tests run using the framework would be able to call for a wide range of specific data or operations from the MTX, and process it in a friendlier and more powerful language instead of Structured Text and other PLC programing languages or CPL, mentioned above, which is a proprietary language of the MTX used together with G-code.

In order to provide simplicity for the users, a class structure was planned in the framework to interact with the services and also to implement helper methods to use NCS functions. To provide better understanding of the service architecture a diagram describing it is shown in Figure 3.

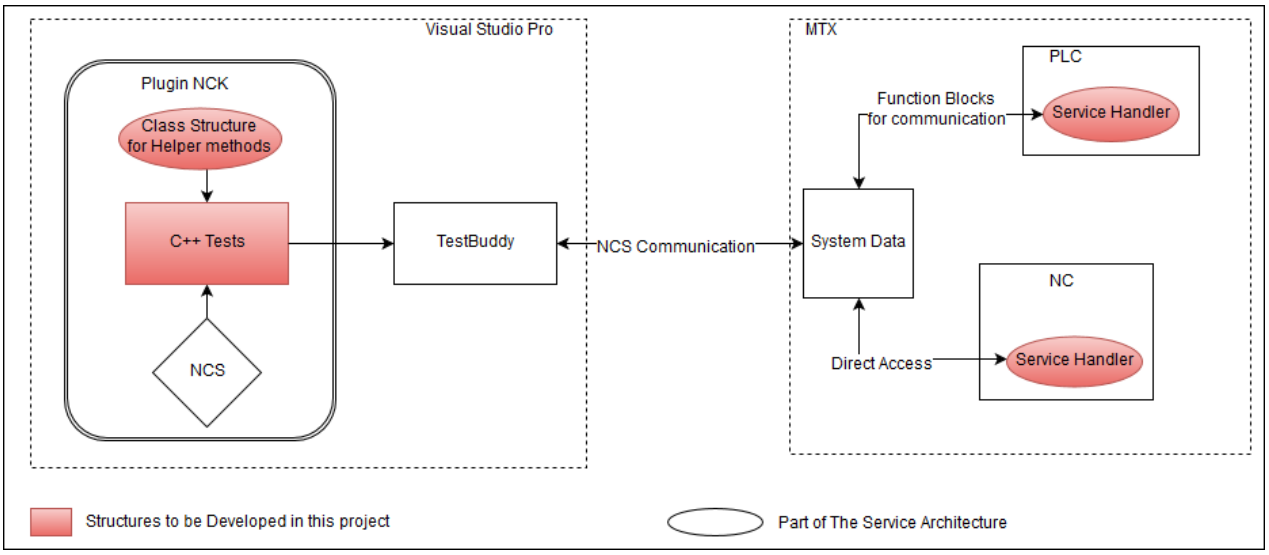

Figure 3 - Service Architecture Description

As mentioned before, the framework was developed in C++ which is usually the language used in the test cases implemented for other systems and so on. But since the creation of the Lua plugin, which enables the framework to process Lua scripts, it became of interest for this project to compare both languages and determine which would be better employed. In this aspect, it was defined that Lua could represent a positive simplifying tool when implementing test cases and therefore, it was also part of this project to enhance the framework to use the service architecture through Lua. Then, evaluate its advantages or disadvantages compared with C++.

To compare both languages, it would be necessary to develop wrapper functions that would provide the Lua scripts with the ability to invoke the methods in C++ from the project. The Lua scripts would work as shown in the diagram in Figure 4.

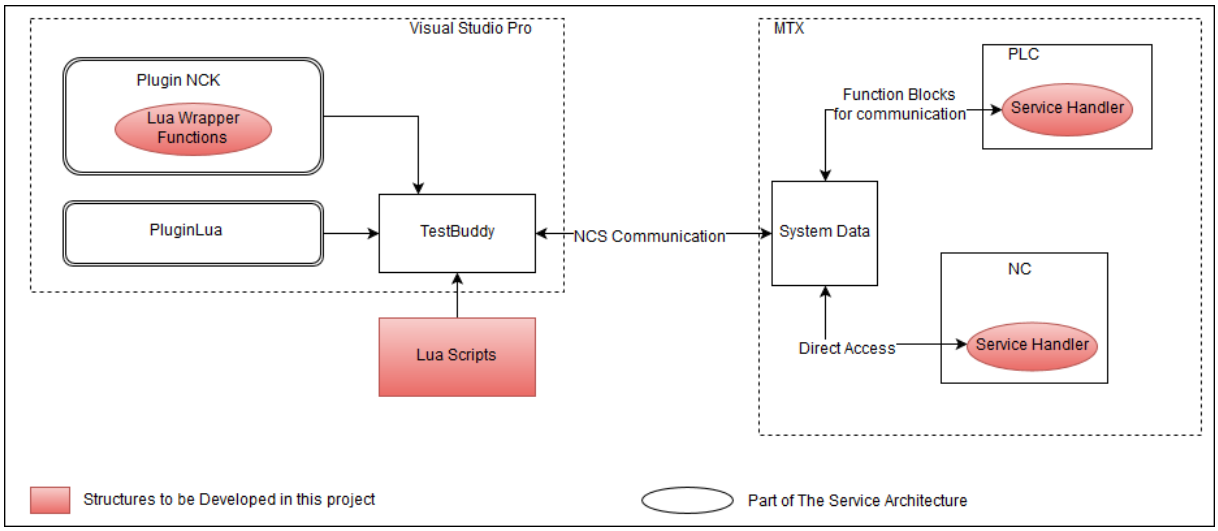

Figure 4 - Service Architecture with Lua

1.3: Services and Tests.

After developing the structure, it would be possible to start concentrating on developing the services and tests. A list of priority tests was made, and from it their necessary services were estimated. In order to test the service architecture, some test cases which needed specific services were implemented and the results refined and validated. This priority list was taken mostly from a second list of system tests already existing in the department, and if there was enough time and the right resources the idea was to implement them all at the end.

For the development of the automatic tests, as mentioned before, the framework would be able to work with C++ and Lua scripts. The later was a language created with the intent to be integrated with other languages easily. This feat is one of its greatest advantages over other interpreted languages and it is accomplished through wrapping. That is a method of programing functions to be used in another way, sometimes just to change the interface, other times just for convenience, but as in this case, it can also be used to integrate two different languages.

It works both ways, allowing other languages to be wrapped in Lua so that their functions can be used in the scripts, or Lua can be wrapped in other languages, so that the scripts can be run from inside the code. For this project the first case was

the one used, where the objective was to use the C++ functions inside Lua scripts. With this method, all the structures that would be needed from the project or other C++ libraries could be invoked in Lua as long as they were wrapped properly.

Although this method gives Lua extreme flexibility to be used together with other languages, as can be seen in [2], the wrapping process can be time consuming and present some complexity. Therefore, in order to access which of these languages was better suited to the project, it was decided to choose a single test case, with intermediate complexity, which would be implemented in C++. Then wrap only the necessary functions it needed and implement the same test in Lua.

The comparison between both languages would then be based on which of them was friendlier, considering that the average users would not be expert programmers. Then the team could choose the language to be used, based on the evaluation performed, using as primary input the conclusions reached by this project.

1.4: Equipment.

During the development of the project, it was necessary to have direct access to the PLC and NC functionalities at all times and to be able to change code and debug at will. To allow such necessities, a test rack was used, composed of one L65 controller, which is a specific version of the MTX, containing a Sercos III device and in this case also three extra I/O ports. This test equipment was also equipped with two simple drives, each capable of working with one actuator and one double drive capable of operating with two actuators. The test bench was also equipped with four actuators, one to represent a spindle and the other three to represent the X, Y and Z axis. This equipment can be seen in Figure 5 and Figure 6.

Figure 5 - Test Rack with two single drives in the black rectangle, one double drive in the red rectangle and four axis attached to motors to represent X axis, Y axis Z axis and Spindle, from left to right respectively.

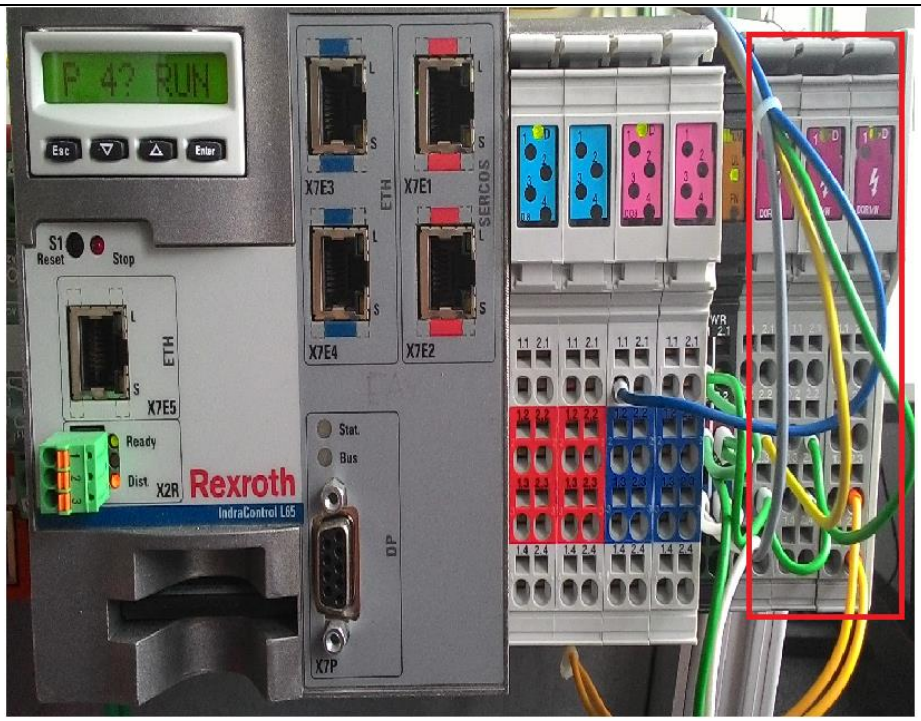

Figure 6 - MTX L65 Controler with three extra I/O ports in the red rectangle

Finally, to be able to force some situations that would normally depend on external conditions, the test rack contained one relay that is connected to the second drive and enables the PLC to turn off the device through an IO port, simulating a turn off. It also contains two communication repeater links that could be activated or deactivated through the IOs as well, making it possible to simulate cable breaks in the communication. These components can be seen below on Figure 7, and in Figure 8 a block diagram that explains the system connections and structure can be seen.

Figure 7 - Signal Transmitters used to Simulate Cable Breaks and Relay used to turn Devices on/off

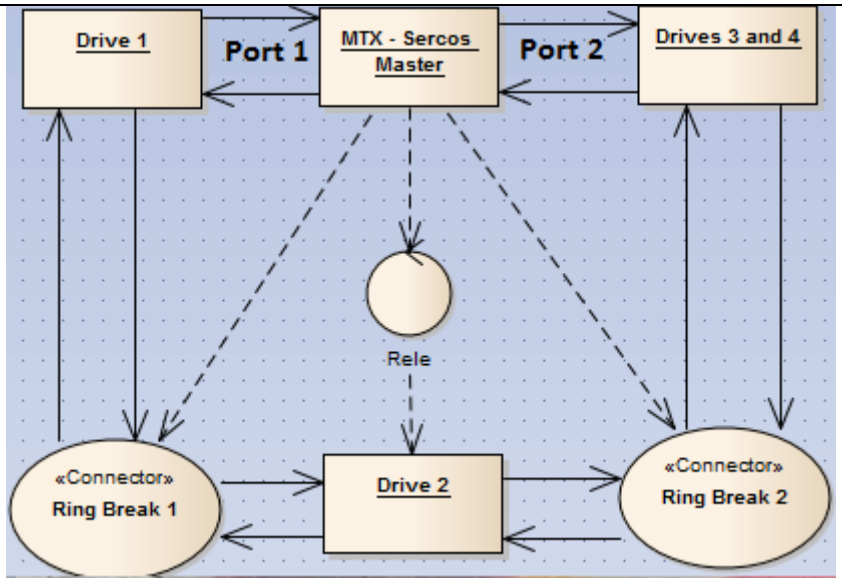

Figure 8 - Sercos Ring Topology with relay and cable break simulators

Chapter 2: The Systems and their integration.

2.1: Systems and interfaces.

In this project, it is not easy for an unfamiliar reader to see how things are connected because they may depend on the interaction of three systems, with code in 4 different programing languages used in more than three programing environments and working not only with the computer file system, but also with the MTX's NC file system.

The PLC codes and functionalities were developed in Structured Text using the software IndraWorks Engineering [3], which is used to perform all the configurations regarding the MTX, its modules, drives, axis, spindles, and in short all the possible configurations of the system. Through this software it is possible to access the NC data structure and modify its archives. It also enables direct access to parameters of the drives and sercos devices.

The coding for the NC is stored in its own file system, in text files with the ".npg" termination. They can be accessed through the IWE when the MTX is connected to the computer, can be created in any simple text editor and then be imported through the IWE, or they can be created directly from the company's program. There is also a third method to access and modify these programs, using a machine interface simulator program, also from Rexroth's portfolio named Indraworks Operation [3]. This program behaves as the display in the real machine would, and allows the user to manually access the MTX functionalities.

The TestBuddy, as mentioned in the previous chapter, is implemented in the Visual Studio Professional IDE. This is actually not only a matter of choice, the VS Pro 2012 provides a library upon which the development of the framework was based. This library is not accessible in the Express version of the same software which forces the development of new features to be using VS Pro 2012 independently of regards to preference or performance of any other IDE's. Even though the teams are already working to change this reality and migrate the

TestBuddy to a version in which there is no attachments to this library, for this project this dependency was still present.

Lastly, in order to create and initialize variables in the System Data from the NC, it was also necessary to develop some XML code, which was done in a standard text editor with support to that language, the Notepad++. The XML files must be located in specific directories inside the NC data structure that can be found in [4]. A graphic description of the systems interaction and its languages can be seen in the diagram presented in Figure 9 in order to simplify the situation.

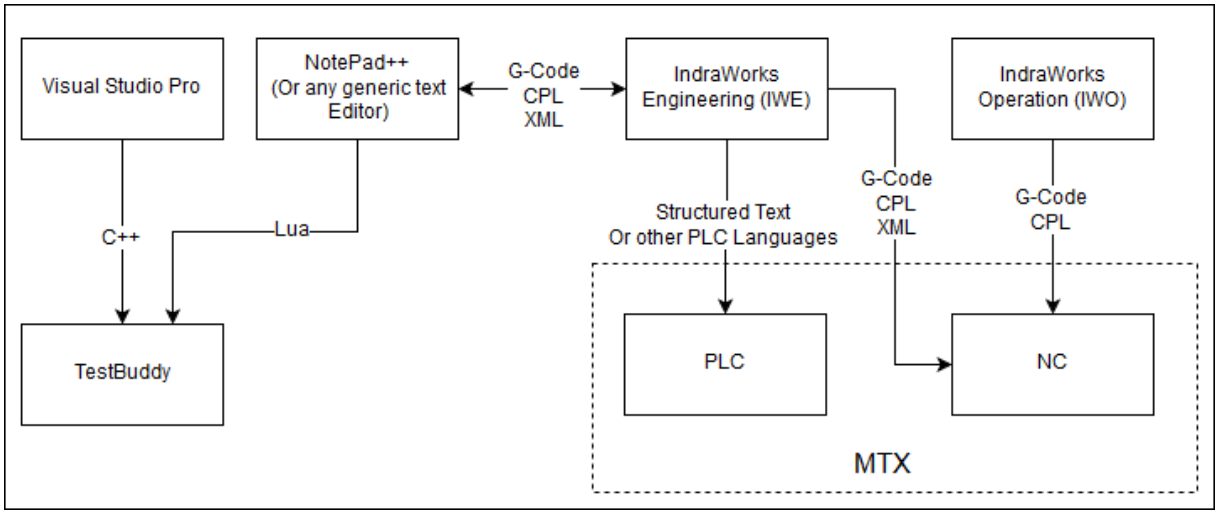

Figure 9 - Systems and Languages

2.2: Sercos.

To fully comprehend the project objectives the reader also needs an explanation of Sercos III in more detail. As mentioned before, the Sercos system is a communication device designed to guarantee real time constraints using Ethernet. It is also designed to provide communication redundancy and robustness. The Sercos communication bus can be implemented in three different topologies, “Simple Line”, “Double Line” and “Ring”. Examples of these topologies are described in Figure 10, using as an example the configuration employed in this project with two simple drives and a double drive.

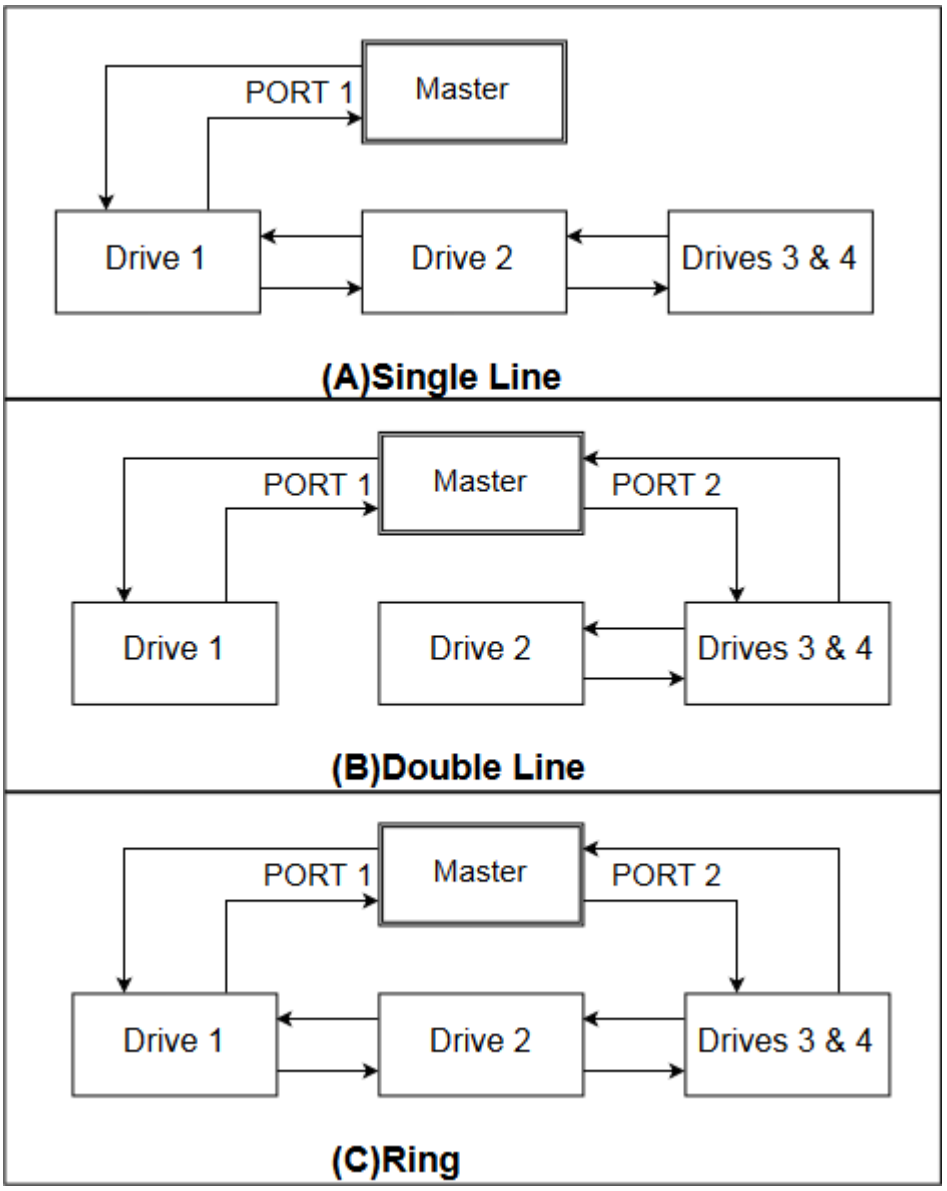

Figure 10 - Sercos Topologies

The system has two Ethernet ports and can work using both when working in Double Line and Ring topologies as seen in (B) and (C) from figure Figure 10, or can use only one in the case of the Simple Line as shown in (A). The communication is established through telegrams sent by the Sercos device which is the “Master” and each device in the topology, the “Slaves”, access their own reserved space for sending data. The telegrams are divided in pairs of one Master Data Telegram (MDT), associated with an Acknowledge Telegram (AT) used by the slaves. Depending on the number of slaves in the topology and the restrictions of data size,

it may be necessary to employ multiple MDT and AT pairs to be able to communicate with all the devices, the higher the number of pairs, the slower the communication restrictions will be, therefore for hard real time restrictions the number of devices has to be taken into account with care.

The most reliable topology for the system is the Ring, where the master sends telegrams from each of its ports in different directions of the ring, establishing a “double” ring, as shown in (C). This way if everything is working properly, the telegrams sent by different ports should be equal when they are received back. If they are different, the system can identify Ring breaks and become a Double Line or Single Line depending on the position of the break, and it would still be able to communicate with all the slaves. This change in topology is possible because the master would establish simple loop back rings for each line, as shown in (A) and (B) where the telegrams reach the last slave and are sent back. Not only that, but the system would be able to identify the position where the ring was broken by comparing the telegrams. On the other hand, if the topology is already in Double or Simple Line and a break occurs, all the devices after the break position are lost.

The Sercos communication bus is also capable of accessing non cyclic parameters from the drives and the system configuration using service channels from the MTX. These parameters can be requested by the PLC, the NC and even by external sources like the TestBuddy framework. Using them makes it possible to access the configuration status and sometimes real time information that might be useful for the tests and, maybe, even for operating purposes.

2.3: System Data Access

The first step to start developing the proposed service structure mentioned in the introduction chapter was to set up the communication methods in the three systems. Each one of them treated the access to the system data in a different way and, therefore, will be explained separately.

2.3.1: PLC

The access to the system data in the PLC is implemented through two function blocks contained in the library from IWE. This two function blocks are called MT_SD_Rd [5] and MT_SD_Wr [5], for reading and writing respectively.

Before explaining how the two blocks work, it is interesting to point some characteristics of function blocks and some differences on its workings compared to normal programing functions in order to further clarify how they were used and why some measures had to be taken to ensure their desired behavior.

One important difference between a function block and a function, in the MTX PLC, is that the PLC's function variables are always reassigned for each call, this way every time the function is invoked with the same inputs, it will return the same outputs, because of that, functions are not allowed to work with variables outside of their scope, accessing global variables for instance. That notion is important because differently from a function, in which the outputs can be accessed only once after the function is processed and then they are reset, the function blocks keep the variables stored and they are not automatically reset. Function blocks also don't have the restriction of using variables of other scopes as inputs or outputs.

Another important characteristic is the fact that the function blocks are instantiable and behave in a way similar to a struct, which can have their contained values accessed by using the notation "<instance_name>.<variable_name>", the only difference being that the output variables from the function block instances can't be written to, only read.

With this in mind is important to remember that the PLC architecture is based on a main loop that repeats itself endlessly and in a really short time. This way the operations performed can't hold the processor for too long or there is a risk of taking longer than the pre-defined loop time, which must be respected. Because of that fact it is common to develop function blocks divided in internal states that may take more than one loop to be executed.

This two facts put together, the capacity to store old values and the existence of internal sub-states to deal with the function execution over multiple loops, represent a potential risk because since the function blocks have the values stored

and take more than one loop to complete, it is relatively easy to make logic mistakes and access the variables at the wrong times, therefore, fetching the wrong values.

To help with this, the two function blocks have an internal Boolean variable called “Active”, as can be seen in Figure 11, that indicates if the function block is still active or not. This helped solving the problem when a read or write operation was executed once. But that variable was not enough when many of these operations were put in sequence, reusing the same instance of the function block.

After debugging and testing, a theory was defined to try explaining the reason for that shortcoming. It was that the execution of the block does not happen at the moment of the call, but instead at the end of the main loop from the PLC, what was evaluated by using the step by step functionality of the debugger. Also, it was observed that the values written or read, when in sequence, were only correct for the first block. This happened because on the following calls, the block already possessed values and therefore they were fetched directly. They were of course wrong, as can be seen below in Figure 12, but the only way to verify this in the code would be to know exactly what values were expected in every situation, which would take all the flexibility from the structure and make it very hard to expand.

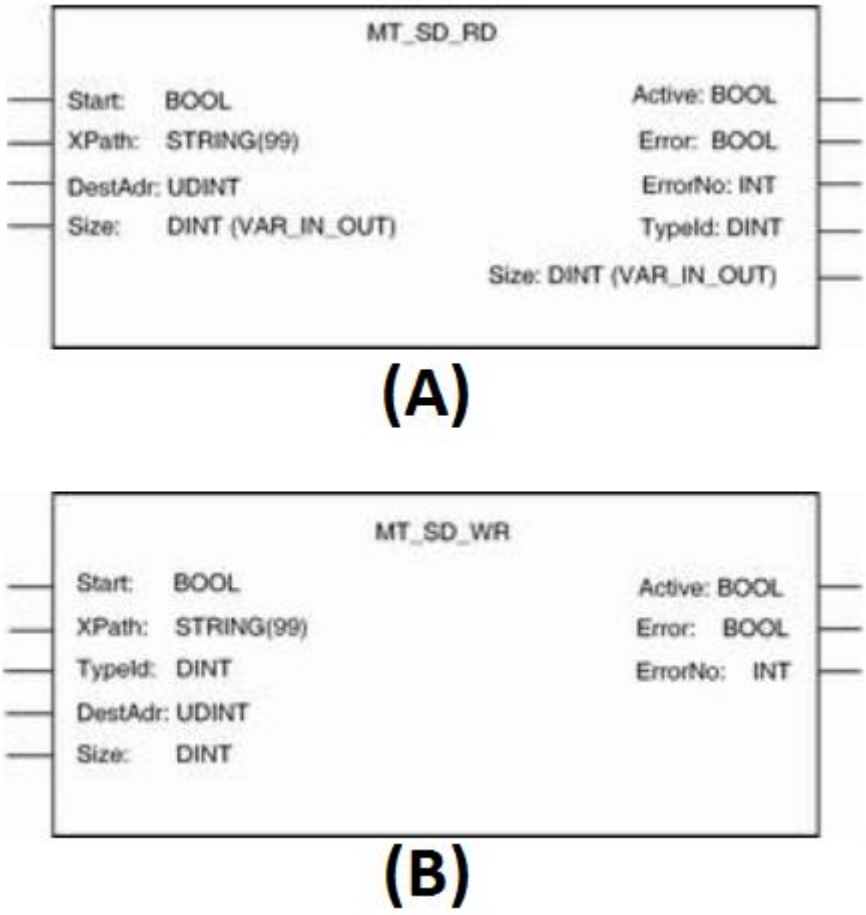

Figure 11 - (A)Function Block MT_SD_RD;(B) Function Block MT_SD_WR

To overcome this obstacle, one other Boolean variable was declared and used to interlock the reading and writing operations using an integer variable to define states, so that they were forced to happen in a sequential manner. This structure will be further explained in the future section regarding the state machine for treating services.

The variables of the blocks are explained in the list below:

- **Start** – Boolean variable used to indicate that the function block must be activated if it is available.
- **XPath** – Array of characters with the xpath in the XML structure for the desired data to be written or read.

- **DestAdr** – Pointer to the address of the data structure where the data must be stored or from where it must be read.
- **Active** – Boolean that indicates if the function block is available or busy.
- **Error** – Boolean value that returns if the function block encountered any errors in the previous operation.
- **ErrorNo** – Integer that identifies the error codes.
- **TypeId** – One of the most important variables of the function. This variable stores the type of the variable stored in the System Data, it contains a number that represents the type in the XML structure and it is not possible to know it without performing a read operation to identify this value.
- **Size** – Integer with the size in bytes of the data.

It is especially important for the operation of the function blocks to understand that since the type for the variables is defined only in the system data, at least one read operation for each type that will be used must be done in order to be capable of performing write operations with those variables.

The function block operation is activated when the start variable is set to “TRUE” and the “Active” is “FALSE”. On the function call, in structured text language, the syntax is as follows:

<MT_SD_Rd_instance_name>(Start:= , XPath:= , DestAdr:= , Size:= , Active=> ,Error=>, ErrorNo=>, TypeId=>);

In this notation, the “:=” operator is used to assign inputs and “=>” is used to fetch the output values. The operation of these blocks can be confusing because it fetches the outputs that are contained in the function block instance at the moment the call is made.

However, the function is only executed afterwards so to fetch the real output values a new function call is necessary. When the function block is called only to fetch the outputs, it must be called with the start value set to “FALSE”, otherwise the function will be processed again without purpose. To make it clearer, in Figure 12 there is a diagram explanation of how a read operation should be executed.

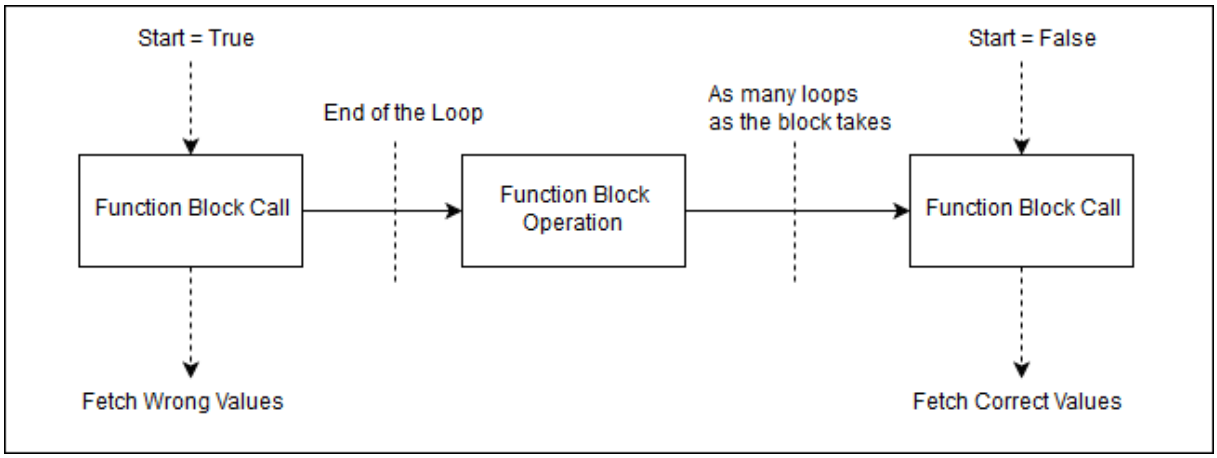

Figure 12 – Function Block Operation Diagram

The user can also access the parameters of the instance directly instead of fetching them, but in order to make the process simpler to visualize and reduce the chance of using wrong values, the option presented above was chosen.

2.3.2: TestBuddy

In TestBuddy, at first, two functions from the NCS libraries open to the MTX client users were chosen to be tested. They were called Ncs_SDSetData[] and Ncs_SDGetData[]. These functions proved to be a challenge to handle and use because they treated all inputs and outputs as XML strings, including the tags needed in the XML structure. Using these two functions made necessary to be able to correctly parse and convert the types to write and read, adding unnecessary complexity to the code, which could prove to be a possible constraint in the future.

Therefore, the approach was changed and it was decided to try the communication with two other functions from the NCS libraries restricted to developers, Ncs_SDSetDataBin[] and Ncs_SDGetDataBin[]. These two functions behave differently from the previous ones because instead of XML strings they use the binary data of the variables directly and therefore are able to process those as any type accepted in the system data. The types that can be used are displayed in Table 1 taken from [4].

Name	Length in bytes	Description	C-Type	IndraLogic type	CPL
string	2*maxLength+1	String(UTF-8 format)	char[]	STRING()	STRING
isoLating1String	maxLength+1	String(Latin 1 format)	char[]	STRING()	STRING
Byte_t	1	Signed 8 bit integer	char	SINT	INT
Short_t	2	Signed 16 bit integer	short	INT	INT
Int_t	4	Signed 32 bit integer	int	DINT	INT
Unsigned-Byte_t	1	Unsigned 8 bit integer	unsigned char	USINT	INT
Unsigned-Short_t	2	Unsigned 16 bit integer	unsigned short	UINT	INT
Unsigned-dInt_t	4	Unsigned 32 bit integer	unsigned int	UDINT	INT
Float_t	4	32 bit real	float	REAL	REAL
Double_t	8	54 bit real	double	LREAL	DOUBLE
Boolean_t	1	true, false, 1, 0	char	BOOL	BOOLEAN

Table 1 - System Data Basic Types

Using these functions gave the program more flexibility but also increased the need for caution, because since they operate with the binary data, it becomes critical to pay close attention to the types and size of the variables that are written or read.

The binary functions work in a similar way to the PLC function blocks mentioned in the PLC communication section, except by the fact that they are executed on the moment of the call and are not instantiable. The syntax for them is as follows.

*Ncs_SDGetDataBin(int UserId, char *XPath_p, void *Data_p, unsigned int *DataSize, int Mode, int *TypeId, int *BasicType, int *Error_p, char *ErrInfo_p, int *ErrorInfoSize_p);*

*Ncs_SDSetDataBin(int UserId, char *XPath_p, void *Data_p, unsigned int DataSize, int Mode, int TypeId, int *Error_p, char *ErrInfo_p, unsigned int *ErrInfoSize_p);*

The inputs for these functions are similar to the presented in the PLC section as mentioned previously and are showed below.

- **UserId** – Integer value with three possible values defined with NCS_SD_MMI_PROCESS_ID_C, NCS_SD_CPL_PROCESS_ID_C and NCS_SD_PLC_PROCESS_ID_C. Usually the last is used but there is not much difference in the function operation for the purpose of this project.
- **XPath_p** – Pointer to an array of chars with the correct XPath to the XML data structure.
- **Data_p** - Pointer to the address of the data structure where the data must be stored or from where it must be read.
- **DataSize** – Pointer to the variable where the size must be stored in case of a read operation, or the value itself when doing a write operation.
- **Mode –** This is an integer with the mode for the operation but for this project it is always mode “0”.
- **TypeId** – One of the most important variables of the function. This variable stores the type of the variable stored in the System Data, it contains a number that represents the type in the XML structure and it is not possible to know in the TestBuddy without performing a read operation to identify this value.
- **SDError_p** – Error code from the system data, it is important to notice this error indicates an error in the system data and not in the NCS function itself since the error from the NCS function is given by the return value of this function which is an “int”.
- **ErrInfo_p** – A character array containing the error message if it exists.
- **ErrInfoSize_p** – Integer that denotes the size of the error string in bytes.

The problems presented by these functions are extremely simpler than the obstacles presented in the PLC part with the function blocks. In the testbuddy it is only necessary to pay attention to the types and when to use pointers, considering the read or write operations. The necessity to perform read operations before write operations is also present in order to define the “Type” value of the data to be accessed.

2.3.3: NC

Finally, the NC part of the communication was the most straightforward of the three because the variables are all declared in the NC file system and therefore can be accessed directly without the use of any special functions. To access the system data variables either for reading or writing one must only use a special notation, similar to the one used for XML XPaths, which are special types of path used in XML structures to find and access information. The difference in this case is that instead of a slash (“/”) as is the standard in XPaths to access substructures, the point (“.”) character was used as if in a struct from C. To exemplify the syntax, considering that there is a structure in the system data such as the one presented in Figure 13 named “CN1”.

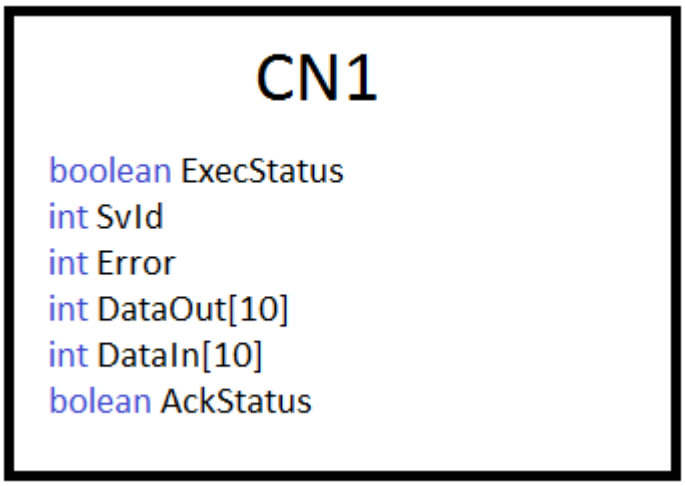

Figure 13 - Channel Data Struct

Then, the access to its variables could be done as follows:

- ***"SD.CN1.ExecStatus"*** – To access the Boolean variable ExecStatus, responsible to signal when there is a service request.
- ***"SD.CN1.SvId"*** – To access the integer variable SvId
- ***"SD.CN1.DataOut.Data[]"*** – To access the individual parameters in the DataOut array. In this case, the initial position of the array is position 1, but when read in the C++ code it becomes position 0.
- ***"SD.CN1.DataIn.Data[]"*** – To access the individual parameters in the DataOut array. In this case, the initial position of the array is position 1, but when read in the C++ code it becomes position 0.
- ***"SD.CN1.AckStatus"*** – To access the Boolean variable AckStatus, responsible to signal when a service has finished being treated.

2.4: Communication

After defining and understanding how each of the systems accessed the system data, in order to enable communication, a complex data type Struct was defined to simulate a communication channel. This structure would contain multiple simple variables that would help coordinate the information between the systems. This complex type was named "CN_t" and was exactly as the "CN1", which is an instance of "CN_t", presented above in Figure 13.

This structure type could be composed from any types available from the system data basic types, which can be seen in Table 1 showed previously. It could also use complex types, which is the case for the two arrays DataIn and DataOut. Arrays are not usually defined as complex types, but for the specific case of being declared inside another structure, in order to simplify their operation in the NC, declaring them in this specific way was the better solution. The difference between complex types to simple types is basically the option to contain elements and parameters, complex types can be composed of simple types or other complex types with no limitations.

In the list below all the variables from “CN_t” will be explained.

- **ExecStatus**: This Boolean variable is used as a flag to indicate to the service providers that a service was requested.
- **SvId**: This integer variable is used to identify which service is being requested by the client.
- **DataOut**: This array of integers is used to send information back to the client.
- **DataIn**: This this array of integers is used to send input information for the providers.
- **Error**: This integer variable is used to send back any information about errors that happened with the requested service.
- **AckStatus**: This Boolean variable is used as a flag to signal to the client that the requested service was processed.

Chapter 3: Service Architecture and Class Structure

Once the communication procedures were defined and the variables could be accessed from all the necessary components, it became possible to develop the service architecture. The first action, before starting the actual programing was to discuss at length the requirements. In the end, this led to some characteristics considered important, scalability, reliability and simplicity to reduce the burden on the users. The first was defined as crucial because at the start of the project, not all the needed services were known. Therefore, having the ability to easily add new services, if the need presented itself, was critical. The second requirement refers to error handling and debugging as well as reliability. The point is that, for the system tests, it is very important to have consistent results, and if they are not consistent the structure must be able to detect where the problem is and handle the error in order to provide useful output information for the users to debug the situation. The third is one of the objectives of the project, to reduce the complexity to work with the tests.

After the requirements, it was important to define the communication protocol using the system data. Considering that all the three systems would access the same variables, a communication sequence was defined in order to synchronize the operations from the client and the servers. This sequence can be seen in the communication diagram presented in Figure 14.

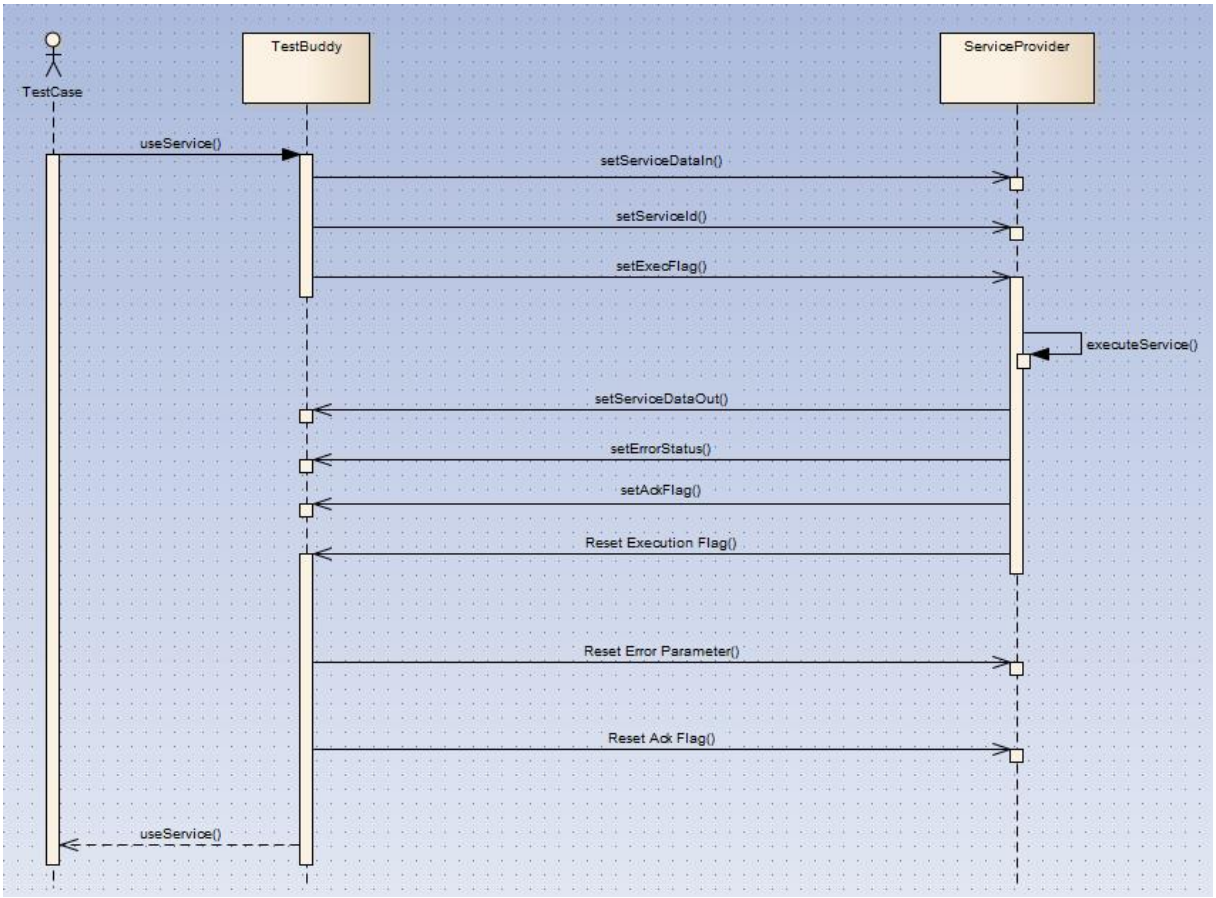

Figure 14 - Communication Sequence Diagram

As can be seen in the diagram, the communication would be centered on the operation of the “ExecFlag” and the “AckStatus” flags. The first would be set by the client to signal a service request, and would be reset by the provider once it was processed. The second, would be set by the provider also when it finished processing the request and reset by the client when it received the service output.

Before setting the execution flag and issue a service request, the client would need to set the data information required by the service and also the service Id. Then the server would execute the service, set the output data, the error status and finally the acknowledge flag as well as reset the execution flag in this order. The client then would finally get the information it needed and then reset the acknowledge flag.

Following the process as mentioned above requires the System Data variables to be in a specific initial state, more specifically both flags must be false.

That is necessary because the order of the operations is important, and if one of the flags is not in the correct state before the operation starts, it will compromise the procedure. This requirement is not a problem since it is possible to define initial states for every variable of the system data through an XML file contained in the NC file structure. In this file the variables can be initialized with a value different than zero or null. The ones that are not referred to are automatically set to zero or null by default. This process happens in every MTX's startup.

To control which services would be executed by the PLC or the NC, ensuring that they could use the same channels without conflict or service identification problems, the strategy used was to make all the service ids present in both providers. All services would be "executed" by both of them, but one of them would always process it as a "do nothing" while the other would execute the real operations. This was necessary because identifying unknown service ids and triggering the right error, is important in both providers. Checking only the Ids which each provider would be responsible for would make them treat the services of the other system as "unknown". In that case, it would be impossible to share the same channels or the error for "unknown services" would be triggered in every service request.

Since the service architecture would be distributed, there were many pieces of the architecture that needed to be developed separately in different languages and systems, as seen in the previous chapters. To better describe the development, a list of these parts is shown below to give the reader a better context. Afterwards each of the topics in the list will be detailed individually.

- **System Data: Data Structures for communication.**

The system data part was to define the data structures that would play the part of "communication channels". It was decided to create a complex data type struct, containing all the needed variables in order to be able to create as many channels as needed easily just by instantiating the struct.

In the end of the project it was also necessary to declare a simple type string in the system data to use as a mechanism to identify the version of the configuration files in the system and provide the right initialization for each test case. This however

is not used by the service structure directly but instead by the TestBuddy to verify the conditions for the tests and is further detailed in Chapter 4:.

- **PLC: Service handler to process service requests.**

For the PLC service handler it was decided to use a state machine approach. This conclusion was reached because not only this way it would be easier to make the system scalable but it also works well with the PLC cyclic loop behaviour. The idea was to divide the states in a way which would completely separate the communication and execution. This way, to add new services only the execution part would need to be enhanced.

- **NC: Service Handler to process service requests.**

For the NC service handler the solutions were limited by the CPL language. Because of that, there were many limitations imposed to the code but since the NC has direct access to the System Data, it was possible to simplify the structure to a simple switch case structure inside an infinite while loop. Each case in the structure would represent a service directly.

- **TestBuddy: Classes to configure communication and to provide support in the creation of tests.**

In the TestBuddy part, the services should be requested through helper functions in order to reduce complexity for the user; otherwise they would need to manually operate the System Data accesses, which would be less than ideal. For this purpose, two classes were created in which the helper methods were defined allowing the test cases to invoke them. The class structure created in the TestBuddy would not only provide helper methods to simplify the service requests but also to simplify the operation of NCS functions and provide aid in the creation of tests.

3.1: System Data

A user can change how the system data is defined through 3 types of XML files. They are, definition files, where new types can be created, declaration files, where the variables can be declared and therefore begin to exist into the system data, and lastly initialization files in which the initial value for the variables can be set.

Both the Initialization files and the Definition files are optional, if the first does not exist the system initializes all variables with a standard value, null or 0 (zero). The second is not needed if the user wants to work only with types that are already defined in the system.

These 3 files can be positioned in 3 different locations inside the NC file system and the repositories are treated regarding a priority scheme. Therefore, if files that define, declare, or initialize the same variables exist, the priority will be given according to the repository where the file is contained, and only the one with the higher status will be considered. The priority is only important considering the files' types, so operations that are done in the initialization files will be compared only with other initialization files and likewise with the other types.

For this project, there was no need for an initialization file regarding the channels, since their desired initial state is the default. The other two files however, were put in the following locations:

- **Declaration File**: "/" - or what is known as the root directory in the NC file system.
- **Definition File**: "/schemas"

The XML Files that were used for the project are contained in the appendix A and if the reader wants to further inspect the syntax and workings of the system data variables, a tutorial on how to use them is presented in [4]. The two directories chosen to put the files in were also defined following instructions contained in that manual.

3.2: PLC – Service Handler

As mentioned before, the service provider in the PLC was implemented in the format of a state machine in order to provide an organized and easily scalable code structure. To achieve this end result, the first step was to define the concept behind the state machine, establish what needed to be done, in which order, and divide those operations in states. After some discussion, the result was a state machine composed by five (5) states, "Initialization", "Check Execution Flag", "Service

Initialization", "Service Execution" and "Service Results" as can be seen in the diagram representation in Figure 15 and in the descriptions presented below.

The state machine is implemented divided in six sub programs, one for each of the states, and one to control their operation and order. This sixth program uses an integer variable to identify each state. During their operation, if the conditions for the next state are met, they change this variable, otherwise they keep the variable with the same value, which will make the selector choose the same program for the next loop until the state meets the conditions.

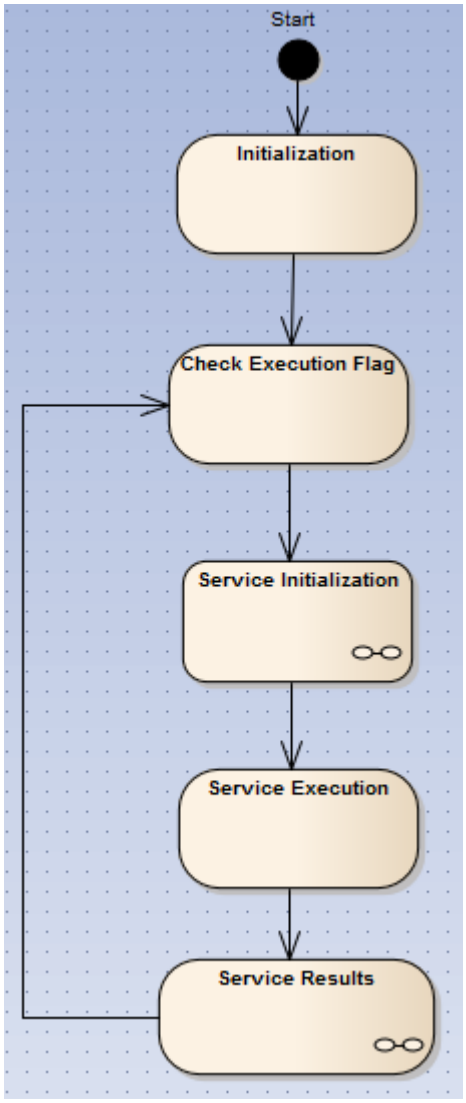

Figure 15 - PLC's State Machine

3.2.1: Initialization

In the PLC's function blocks, for sending and receiving binary data to and from the system data, one of the necessary inputs is the "Type" of the variable, as presented in Chapter 2:. For the read operation, a pointer to a space in the memory where the type must be stored has to be provided, and in the write operation, the actual type number has to be given.

This type is transmitted as a number, available only inside the system data. For this reason, before executing any writing operations in the PLC, at least one read operation must be executed for each specific type. For simple types just one read needs to be executed and then the fetched type may be used for all other variables of the same type, but for the complex types the user has to make sure to fetch the type value for every variable since they may be different.

This first state is responsible to fetch the type for Boolean variables, used in the "ExecStatus" and "AckStatus", the type for integer variables, used in the "SvId" and "Error" and to get the types for the two complex type arrays "DataIn" and "DataOut".

These values are stored in global variables inside the PLC, which are used by all the states to execute write operations. This way the fetching operation only needs to happen once instead of before every write operation. For that reason, this state runs only once before the cyclic operation of the state machine starts.

To simplify the interaction with the channel structs from the system data, a type struct was declared in the PLC to be able to mimic the one in the system data. Its only difference is that since the PLC needs the Xpaths in order to access the right variables in the system, it has one Xpath for each variable, as can be seen in Figure 16. These paths must be initialized when an instance of the struct is declared.

It also contains one variable to control the state of the state machine, it was done this way because then the program can use multiple state machines working with different channels in the same loop, simulating parallel operation, and enhancing the project scalability.

```
TYPE Channel :
    STRUCT
            SvId        : DINT;
            SvDataIn    : ARRAY [0..9]OF DINT;
            SvDataOut   : ARRAY [0..9]OF DINT;
            Error       : DINT;
            ExecStatus      : BOOL;
            AckStatus       : BOOL;
            XPathStat       : STRING(30);
            XPathAck        : STRING(30);
            XPathError      : STRING(30);
            XPathId         : STRING(30);
            XPathDatIn      : STRING(30);
            XPathDatOut     : STRING(30);
            State           : DINT;
    END_STRUCT
END_TYPE
```

Figure 16 - PLC's Channel Struct

3.2.2: Check Execution Flag

The second state is responsible to check if a service request has been made. That is done using a polling strategy, once every 2 loops from the PLC, each with 20ms at most, the flag is checked and if it is recognized as active, the next state is triggered. The reason why it takes two loops is because this state performs a read operation, which takes more than one loop as mentioned in chapter 2.

This state is extremely simple and if the architecture was contained in a single program or environment, it would be easily compared with a conditional operation like an “if”. Its only complexity is the necessity to read the flag from the system data.

3.2.3: Service initialization

The third state is responsible for fetching the service id and the data input array from the system data. The first is used to identify the service that must be performed, the second is not always used but to simplify the logic behind the state machine operation, the array is always copied from the system data and it is left to the execution logic to define if it will be used for that specific service or not.

As mentioned in the previous chapter, in order to cope with the read operations in sequence, this state was divided in sub states, to manage the execution order and guarantee that the right information was read. The states can be seen in Figure 17.

This was necessary because when only the active flag was employed, and the function blocks were in sequence, the fetch operation was controlled to happen if “Active” was not true. But after the first read was done, since the instance of the function block stored the values from the previous read, the following operation would fetch the wrong values immediately at the moment of the call, as showed in Figure 12 from Chapter 2:. Moreover, since this operation would be used for every service, there is no way to know exactly what the expected values are. The option to check if the values were fetched could be at best, as simple as “a value” was read or “no value” was read. This check does not help in this situation, because the fetched value would exist, even though it would be wrong.

The method used to ensure that the right values were read regardless of the fact that the function block was being used in sequence, as mentioned in section 2.3.1:, was to ensure that every time the block was activated, before checking if the values were fetched, the system also checked a Boolean variable called “SecondLoop”. This solution was devised to simply force the read operations to ignore the values fetched at the first call and consider the values from the second loop onwards, with this double check the “Active” variable became sufficient to ensure that new values were read.

Therefore, in each of the sub states with read operations, the SecondLoop variable is used to ensure that the correct data is read.

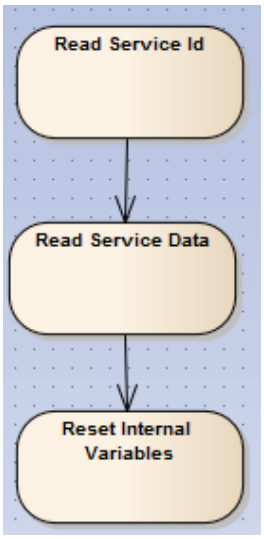

Figure 17 - Service initialization sub states

3.2.4: Service Execution

The fourth state is responsible to identify which service to execute and then do it. The plan for the state machine is based mostly on how to treat this state, because it is the only state responsible for the execution itself. The other states are used to standardize the communication and how the services are treated, and this state is responsible for their execution. This way, if the need for new services arises, in order to implement them, only three (3) steps should be taken.

- **Create a new service Id on the service list to act as an identifier**: the service list is an Enumeration list in which the services can be addressed in a friendlier way for the developers to keep track of the service Id's easily.
- **Create a new program file**: In the PLC, since functions must always be reset by definition and they can't operate with variables from different scopes, their operation is different than what is common in other programing languages as mentioned in section 2.3.1:. In example, because of that restriction, these functions are unable to work with global variables and some other tools.

  Therefore this so called "program" in the PLC is nothing more than a structure which acts as a function would from other programing languages

perspective, like C in which we can define inputs, use pointers, access global variables and so on.

In this structure in order to control if the service has ended or if it is still executing, an output variable called “OUT” was defined in every service, and when it is TRUE the state machine will trigger the next state, otherwise it remains in this state as many loops as necessary to complete the selected program.

- **Create a new case in the switch case structure that identifies the services**: After having a new program that executes the service and an identifier in the list, adding a new case for the structure is as simple as using the new identifier to create the new case that calls the program. Lastly it is necessary to add the necessary handling of the “OUT” variable mentioned above.

Following this logic, every service is executed in its own way, depending on their need, as long as they respect the convention for the output variable for the PLC. The service execution also operates directly with the PLC channel struct presented in Figure 16, it uses the input values fetched in the previous state, process them however it needs, and then it should store the outputs for the TestBuddy in the same structure using the “DataOut” and “Error” variables for the next state to send them to the system data.

3.2.5: Service Results

The last state is responsible to write the outputs achieved in the previous state to the system data, then set the acknowledge flag to true and reset the execution flag, in order for the client to know that the service has been executed. This state always writes all the output variables to the system data even though the service might not have used them. It is the user responsibility to know if the requested service should provide outputs or not and therefore treat them accordingly. It is also the user responsibility to know which error codes are possible for each service since they treat with a huge amount of functions that have their own error code lists and

the codes may overlap with other services. The error handling was done this way, because even though it is on the scope of the project, creating a global index for all the error codes from all the systems that interact together was not.

Finally, this state was also divided in sub states for the same reason as the Service Initialization in 3.2.3:. Since the problem and operation is similar, it will not be described again. In Figure 18 the sub states can be seen.

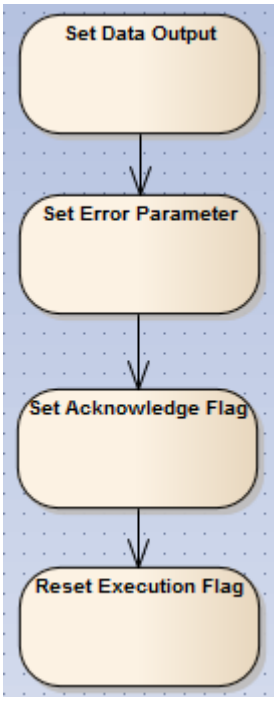

Figure 18 - Service Results Sub States

3.2.6: Service Descriptions

Now that the state machine for the PLC has been explained, a list of the implemented services will be provided with a description of each of them, in total there were 10 services implemented.

3.2.6.1: SetIO – Service Id: 1

This service was created to control the output of the IO ports 1, 2 and 3 denoted previously in Figure 6, which are responsible for the ring break between

drives one and two, as well as the ring break between drives two and three and for turning drive two on/off respectively. The service receives as an input which IO to change, and to what value, 0 or 1. This input is passed as the two first elements of the DataIn array, in which the first element identifies the IO and the second defines the value to be set. The bit position can be one of the three previously mentioned values that are mapped using an enumeration to identify the Boolean variables directly linked to the hardware bits in the PLC. With this direct link, once the variable is set, the real bit is also immediately set.

Since this service acts through a Boolean variable, it could be extended to set the values of common Boolean variables, if the need presented itself but this was out of the scope of the project until this point. The Enumeration that maps the bits is shown below in Figure 19 to exemplify the variables mentioned.

```
TYPE BitMap :
(
    Ring_Break_1    := 1,
    Ring_Break_2    := 2,
    Turn_Off_Drive  := 3
);
END_TYPE
```

Figure 19 - SetIO BitMap

For this service to be used by the TestBuddy, there is an equivalent Enum defined in a header file to match the one above in order to make the service requests easier for the programmers. It is important to keep both enumerations updated in the PLC and in the C++ header in order to avoid complications.

Lastly, there is no output for this service, only the error parameter is set.

3.2.6.2: Service Drive Power On – Service Id: 2

This service also sets an IO bit, but since the behavior is specific, it was decided to make a separate service to solve it. This bit is connected directly to a switch in the power control mechanism that automatically switches the power button

internally and enables the current to flow, allowing the system to be energized by the base voltage of 700V.

This service takes as an input the status for the power, either 1 or 0, however, it is not capable of deactivating the power through this command. Once the power is on, when the bit is set to zero, the test rack stop button still has to be pressed. It is important however to be able to set this bit to zero because if it is 1, and the stop button is released, the system immediately re-energizes and will keep doing so unless the IO bit is set to zero. Without this option the bit needs to be manually set through the IWE debug mode or by restarting the whole system.

Being unable to turn the system off automatically is a restriction imposed by the hardware, and could not be overcome during the project. This service provides no output aside from the Error parameter just like the previous service.

3.2.6.3: Service Check Topology – Service Id: 3

One of the greatest interests in this project is to perform specific tests for Sercos functionalities, and many of them are dependent on the topology used in the communication and how that topology changes in reaction to unexpected situations. This service does not receive any input, but it produces three outputs:

- **Topology**: This can be a value from 0 to 4. This value can mean, Unknown Topology, Single Line Topology, Double Line Topology and Ring Topology respectively.
- **Ring Break**: If there is a ring break, this value returns its position. It starts as 0 in the position between the master's port 1 and the first drive and increases 1 for each drive it passes, therefore i.e. if the break was between drives 1 and 2 the ring break would be 1, if it was between drives 2 and 3 the ring break would be 2 and so on.
- **All Devices**: Returns a Boolean variable that is true if all the devices in the topology are active and false if one or more devices are inactive.

This service uses a specific function block called MT_RingTop [5], defined in the IWE library, since the internal workings of the function are not especially

significant, the reader can find more information on this function in the reference manual if there is any need.

3.2.6.4: Service Set Override – Service Id: 4

This service is responsible to set the override value for the axis and the spindle drives. It is necessary for tests that run NC programs which do G code operations with the axis and spindles to move. If the override is not set, it is defined as 0% and then all speeds defined in the G code will have 0% of the values defined in the code. Usually for these tests, the interest is to set the override to 100% both for the axes and spindles, to simplify the operation of the G code programs.

The service uses the internal structure of a virtual simulator from a control panel called VAM, already defined in the PLC program, to change the override. Because of that, there are 16 possible values, the same used by the VAM.

For the Axis the values correspond to: 0%; 1%; 2%; 4%; 6%; 8%; 10%; 20%; 30%; 40%; 50%; 60%; 70%; 80%; 100%; 120%.

For the Spindle however they are: 45%; 50%; 55%; 60%; 65%; 70%; 75%; 80%; 85%; 90%; 95%; 100%, 105%; 110%, 115%; 120%.

At this point this service is only interested in setting both overrides to 100% but it could be easily modified to provide finer configuration options.

This service has to use the variables that were already defined for the VAM that range from 0 to 15 since the switch in the virtual panel has 16 positions, each position is translated to the system as a byte and the order is not crescent so the service accepts the numbers 0 to 15 and performs the conversion to the byte values for the VAM in order to be simpler to the user. The conversion is done using Table 2.

Switch Position	Byte Value	Axis Override	Spindle Override
0	0	0%	45%
1	1	1%	50%
2	3	2%	55%

3	2	4%	60%
4	6	6%	65%
5	7	8%	70%
6	5	10%	75%
7	4	20%	80%
8	12	30%	85%
9	13	40%	90%
10	15	50%	95%
11	14	60%	100%
12	10	70%	105%
13	11	80%	110%
14	9	100%	115%
15	8	120%	120%

Table 2 - Switch and Override values

3.2.6.5: Service Set Drive Ready – Service Id: 5

This service was created to enable the drives for operation. In that respect, just being energized is not enough, the drives must be set to the ready state. The service alters the drive ready state by using the VAM internal structure, just like the previous service.

However, internally the VAM treats this process differently. When the trigger is pressed, the VAM toggles the state based on a rising edge of the button, which is connected to a Boolean variable. Therefore, when the user invokes this service to set the Boolean variable on, it must remember to set it off afterwards. This implementation was preferred because if the service reset the variable internally, there was no measure of how long the variable would need or how many loops, in order to be properly processed by the internal structure of the VAM. But it is known

that the time and number of loops to invoke the service through the system data would be sufficiently big to ensure the VAM processing since the difference in scale is significant.

The input for the service is the state for the trigger Boolean variable to simulate the rising edge; this is done automatically by the helper methods in the class structure presented further for the TestBuddy. There are no outputs from this service only the error parameter.

If the drives are set to ready state without being energized, the system drives will automatically enter into an error state and the system will need to be restarted. Therefore the operation of this service must be handled with care and before turning the power off, the drives state must be set to “not ready”.

3.2.6.6: Service Nc Stop – Service Id: 6

This service also uses the structure created for the virtual simulator through a Boolean variable. Its objective is to stop a program running in the NC at any given time. Once this service is called, it enables the program stop by setting a Boolean variable responsible for that operation and disables a Boolean variable responsible for the program start operation, causing the program to stop. There is no need to disable this stop variable afterwards because that is done in the Start service described in 3.2.6.7: below. If no error is encountered it returns zero at the error parameter, it has no other outputs.

3.2.6.7: Service Nc Start – Service Id: 7

It is possible to start a NC program through the NCS interface, so in theory there would be no need for this service. However the NCS function needs the NC channel to be in the “Ready” state but after a forced stop for example, the state becomes “Stop” instead of “Ready” and the function does not work. This service was created to solve that need, it works exactly as the previous service but it sets the variable for program start and resets the variable for program stop. This service should be able to start the NC program if the channel is in any state that allows the start of the operations, it does not have any outputs aside from the error.

3.2.6.8: Service SVC Read – Service Id: 8

SVC parameters are responsible for many configurations of the devices connected to the MTX and Sercos. This service enables the TestBuddy to access them and read their values to an unsigned integer. Because of this type restriction, not every parameter can be accessed through this service, but this was not a problem since there are NCS functions that also perform this operations directly. This service takes as an input the number of the SVC parameter, and the number of the device from which to access it. If no error is detected the error parameter is returned as zero, and the value that was read is returned in the first position of the DataOut array.

3.2.6.9: Service SVC Stress – Service Id: 10

This service is different than the previous one because it is used to perform a stress test, it receives the number of the device it has to access, the number of read operations it must perform and the number of the parameter to access. As an output it receives the number of read operations it performed and the error parameter returns the counter of how many access errors occurred, therefore if all the requested accesses were successful the error should be zero, if any other positive value is returned, that would indicate the exact number of failures that occurred.

3.3: NC Service Handler

The NC service handler was implemented in a language called CPL, which is used together with G code and therefore the program is done in a *.npg file. The language provides a range of logic operations and functionalities but has many limitations on how to operate data and also to create structures. With all those limitations in mind, the program was simplified to a switch case structure for each channel that will be used. This structure would be contained in an infinite while loop and would act pretty much the same way as the PLC, first it keeps checking the Exec Status flag, once that is recognized as true, the selector would use the service Id to identify the case and perform the requested service. Since the NC has direct access

to the Sytem Data as mentioned before, the services can write the output directly and set the acknowledge flag.

For the NC, the number of implemented services was a lot lower than in the PLC as the functionalities needed from the NC are less numerous. Moreover, since the NC program is in an .npg file, which does not run by default, in every test that needs the NC handler, the right file must be loaded.

3.3.1: Service NC SVC Read Stress – Service Id: 9

This service implements the same operation as the service described in 3.2.6.9:, except that the execution is done in the NC and because of that reason, the Sercos Parameter can't be changed, as of the time it was implemented, it uses parameter S-0-0380. This difference is due to the limited string operations available in CPL, because the function that performs the SVC access in the NC uses a string to describe the SVC parameter, differently than in the PLC and in NCS which take an integer value to describe it. To overcome this problem could take too long, therefore it was preferred to fix the accessed parameter.

The service takes as input the number of read operations to perform and also the drive id in which to do so. As an output it also returns the number of errors in the read operations, therefore if no errors are detected the error output must be zero, and it uses the first element of the DataOut array to send the number of read operations that were actually performed.

3.3.2: Service Test Mux – Service Id: 12

This service was created specifically for the Mux test. Originally the test would perform some operations and write specific values in restricted memory space, the *.npg program used in the procedure was modified to right in variables from the system data. This service fetches the values of those variables, it does not need inputs and sends two outputs in the DataOut array. If no error is detected the error variable is returned as zero.

3.4: TestBuddy – Class Structure

In order to reduce the complexity of using the service architecture as well as NCS functions, it was decided to create classes with helper methods to enable the necessary operations for the test cases.

The structure of the classes was based on certain functions that were already implemented in another prototype. To manage the connection with the MTX, a NCS function with the name NCS_OPEN was used. This function uses a port and an IP address to establish the communication, as well as some other simple configuration parameters and three more complex parameters that are, a pointer to an “OnExitHandler” function as well as two other pointers, one to the program window and the other related to the object instance. The way to use this function was copied from the prototype, specifically to not worry about these complex parameters that were already solved.

The disconnect operation was also taken from that prototype, because the way to do it depended on the way the connection was established but, also in this case, it relied on a “OnDisconnect” function that terminates the connection when the instance, which is connected, is destroyed. This means that the disconnect operation was performed once the connected object was destroyed.

It was decided to use these functions to reduce the complexity of the project, otherwise establishing the connection could have proven an obstacle, delaying the start of the project. Considering that a prototype was already functional to some extent, it was more interesting to proceed with the implementation.

The first class created for this project was called “NcsInterface”, and the idea was to use instances of the class to establish connections with the MTX. These instances would be used to access the helper methods, working as interfaces.

This solution was preferred instead of using static classes because, as the project progressed, it became important to be able to disconnect and reestablish connection during the same test case. To do that using the “NcsInterface” it was needed to use the OnDisconnect function mentioned before, therefore either the program had to be closed, or as this solution works, the instance of the NcsInterface class could be deleted.

At this point with the NcsInterface class it was possible to establish and reestablish connection in the same test, but that was not done automatically, since a helper method inside a class that declared or deleted an instance of the same class couldn't be created. For this reason, in order to create helper methods to handle these situations regarding connection, a second class was created.

Instead of using inheritance which would provide it with the same methods as the NcsInterface, this new class contains an object of the "NcsInterface" as a public member, this way the new class would control the connection through instantiation and destruction of the "NcsInterface" object, and the functions would still be easily accessible. The name of this new class was defined as "Utilities".

To map the services and some useful information like the bits in the PLC that could be altered, or the reference number of the channel structures in the system data as well as the service ids, a header file was also created with the name of "maps.h", containing the definitions to be used by the helper methods. These are the three parts that form the class structure meant to simplify interaction with the service architecture and some useful NCS functions to the tests. They will be described in further detail below.

3.4.1: NcsInterface Class

As mentioned before, this class was devised to take the complexity from the user, in this case, the person creating new test cases. To do so, a list of the tests was taken, as mentioned in a previous chapter, and the necessary operations for those tests were estimated. In reality they were also corrected or improved during the project implementation when problems were detected, because at first, the conditions were estimated, and sometimes the available functions had limitations that had to be solved. From that, services were derived, but since there are NCS functions that provide specific operations, some were mapped directly to helper methods, without the need for any service.

In the constructor, a connection is stablished with the MTX and some internal variables of the class are initialized. When the instance is destroyed, the destructor handles the connection termination invoking the necessary functions.

In order to handle the service architecture operation, the class possesses some private variables related to the System Data variable types and XPaths, which can be used by every method. There are methods specifically created to manage the initialization of these variables or to manage the XPath strings based on the channel number. Each method will be described individually in order to provide the full scope of the class capabilities.

3.4.1.1: void NcsInterface::InitChan()

This helper function is automatically called in the constructor of the class to simplify the operation for the user, but it can be called at any time during execution as well. This method takes a “Channel”, defined in the “maps.h” header, as input and initializes the types for the NcsInterface instance: TypeInt, TypeBool, TypeDataIn and TypeDataOut. This method is mandatory to ensure that all the other methods which access the System Data work properly. It writes the types directly in private variables from the instance.

3.4.1.2: int NcsInterface::CheckTopology()

This helper function is used to call for the service “Check Topology” described in 3.2.6.3:. As can be seen in that section the service has no inputs and returns three output values, “Topology”, “AllDevices” and “RingBreakPos”, as well as the error status. Therefore, this function requests the service and polls for the results, which it takes and writes in variables provided to the function in the format of three pointers to integers.

At the end, if no errors occurred, the function returns the value zero, otherwise it can return negative values like minus one (-1) for simple errors, or repass the error codes from its internal calls, i.e. the PLC function blocks or the NCS functions used and so on. The function also takes as input which channel in the system data it should use to perform its operations.

3.4.1.3: int NcsInterface::SetIO()

This function was created to use the service described in 3.2.6.1:. As described there, the service takes two inputs, first the bit that must be set and second its value, either 0 or 1. Because of that, these two values are also used as inputs to this helper method. Moreover it also takes as input the system data channel it should use to request the service.

Its operation works by requesting the service with the desired inputs and waiting for the acknowledgement. The service has no output values aside from the error status so this function only takes that value and repasses it as its output. If no error occurs it returns zero.

For both the “Channel” and the “I/O Bits”, there are Enumeration maps defined in the “maps.h” that can be used by the user to simplify its operation.

3.4.1.4: int NcsInterface::ExtractTar()

This method does not use any of the created services, it uses a specific NCS function in order to load configurations extracted from a file in the “*.tar” format to the MTX. That file can be generated through the IWE and contain seven blocks of configurations that can be individually set to be loaded. They may contain information like NC file system configuration files, ram definitions, and many other specific parameters that define how the MTX will work.

This method takes as an input, the full path to the “*.tar” file and an integer for each of the seven configuration blocks that can be set as 0 or 1 to indicate which of them must be loaded. If the configurations are properly loaded the function returns zero as the error output.

3.4.1.5: int NcsInterface::PhaseChange()

This method does not use any service; it takes as input the desired phase and start-up mode, the target device, as well as an internal timeout value. Because of specific details of the NCS function used, there are two possible modes, “topology mode” and “normal mode”. The first can only be used with one device at a time; it is

specific because it enables a change of operation mode, but because this type of operation is too complex, a specific method for mode changing was devised to provide more simplicity for the user. This method can also be used for mode changing, but it requires more knowledge about the NCS function used and also about the internal workings of the MTX.

When the normal mode is used, instead of performing the phase change operation in one specific device, the function does it with all the devices in the topology. When the normal mode is selected, the input for the device must be changed to minus one (-1).

For this function a debug variable was also created. If it is different than zero, it makes the function print its internal status during the operation; otherwise it only prints the final results. If no error is detected, the function returns zero.

3.4.1.6: int NcsInterface::PhaseChange_Simple()

This method is a simple wrapper for the previous one to hide all the complexity from the user. It takes only the desired phase as an input and can only operate with the normal mode setting the other parameters to default values. If no error is detected it returns zero.

3.4.1.7: int NcsInterface::ModeChange_PM()

This method uses the same NCS function from the two previous methods but to simplify its operation the only input in this method is the desired device to be switched. If the function is able to switch modes correctly it returns zero as the error.

The reason to create this method was actually because the function used in these three cases, 3.4.1.5:, 3.4.1.6: and 3.4.1.7:, operates based on the desired phase. However, the system not only has normal phases from 0 to 4 but also two different operating modes, which are OM and PM. The first stands for “Operation Mode” and is the default, the second is a special mode that stands for “Programming Mode” and should be accessible in every phase after phase 1.

The problem is that, instead of creating a separate function to handle the mode changes, the developers included that functionality in the same NCS function. What the function actually does is that when the requested phase is a special value named “4PM” which is represented by an integer of value “20”, instead of switching the phase, it switches the mode.

This NCS function was created this way because at the time, it only made sense for the development team to switch to PM mode in phase 4. However, the system also accepts the command in phases 2 and 3, which makes using the function and verifying the results confusing and more complex than needed.

To make matters worse, the drives display’s only show the PM mode indicator if the switch is done in phase 4. It is possible to verify the mode by checking internal parameters of the drives to confirm the real situation in phases 2 and 3, but the average user might not be aware of this feature and get confused.

3.4.1.8: int NcsInterface::SetOverride()

This method uses the Set Override service described in 3.2.6.4:. Although the service accepts more than one input, for all tests created, there is only interest to set it to 100%, therefore the only needed input for this method is the System Data channel to be used and the service request will be done with the correct parameter for 100%. If the override is correctly set, the function returns zero.

3.4.1.9: int NcsInterface::SetDrvPower()

This method uses the service described in 3.2.6.2:. As the description of the service mentions, the objective is to set the power on or to deactivate the bit so it can be turned off manually. This method takes as input the desired status 0 or 1 and the channel it should use, if the correct bit is set, it returns zero.

3.4.1.10: int NcsInterface::SetDrvReady()

This method uses the service described in 3.2.6.5:. As the description points out, the operation of the VAM, for the used parameter, works in a toggle strategy

triggered by a rising edge. Because of that, after the service is used to set the Boolean variable to “true” it has to be invoked again and set to “false” before it can be used again.

Taking that into consideration, this method invokes the service two times in sequence to work as a toggle, one sets the parameter to “true” giving the VAM the rising edge it needs, and then, it sets it to “false” automatically. This way the user does not need to worry. The only needed input for this function is the system data channel to be used and if no error occurred it returns zero.

3.4.1.11: int NcsInterface::ExecProgSelect()

This method does not use any services, it uses NCS functions directly. It takes as input the channel number from the NC, and it is important to be clear that this channel does not have any link to the ones created through the System Data, this are actually NC operating channels to run NC programs. And it also takes the name of the desired program. It does not take the full path of the program because it searches for it in a specific directory inside the NC file system, with the path “/usr/user”. If the program is loaded to the NC operating channel correctly the function returns zero. This function does not start the program, it only loads it.

3.4.1.12: int NcsInterface::CheckForNcState()

The NC operating channels have status of operation that can be seen in Table 2 taken from [6]. They can be used to determine what is happening in the NC in real time and depending on the situation, can be really useful and simple to help the interaction with the NC.

Meaning	Symbol	Explanation
Channel state The status of the active NC channel is shown as symbol and text in this area.		Ready
		Running
		Stop
		Prepared
		Malfunction
		Inactive
		Control Reset

Table 3 - NC Channel States – Representation in the machine operation panel

This function is used to control the operation regarding NC programs because the TestBuddy has no information of what the program will do unless it was especially created to communicate, as for example the NC program for the service architecture. But for other programs that don't interact, for example programs that just test a motion sequence from the axis or spindles, can only be synchronized with by observing the channel status.

The method does not use any services, it accesses NCS functions directly. The inputs taken are the operating channel number and the desired status identifier. If no error happens, which means to say that, if the channel is in the desired state, the method returns zero, otherwise it returns error codes to inform the situation.

3.4.1.13: int NcsInterface::ExecProgStart()

This method does not use any services; it directly starts the channel operation in the NC through the NCS interface. It's only input is the channel identifier. If the program is successfully started it returns zero.

This function will only start the NC channel if it is in the "ready" state, therefore if a program is stopped halfway through for any reason, and the channel is for instance in the "stop" state, this function will return an error. This restriction is

generated by the NCS function provided, there was no way to overcome it through the NCS with the available functions.

3.4.1.14: int NcsInterface::ProgStart()

Since the previous method was limited regarding the start operation of NC channels, another method was created, this one uses the service described in 3.2.6.7:. Differently from the previous service, which starts the chosen channel, this method starts every available channel that is being used. Its input is only the system data channel to be used. If no error happens it returns zero.

3.4.1.15: int NcsInterface::ExecProgStop()

This method uses the service described in 3.2.6.6:, it receives as an input the system data channel to be used. If no error is performed it returns zero. If the method succeeds, the channel status should become “stop” as seen in Table 2

3.4.1.16: int NcsInterface::ExecChanReset ()

This method does not use services, it receives the desired channel to be reset as an input and through an NCS function resets the channel. Which means that any operation being done in that channels is stopped, the loaded programs are removed and the status when the function finishes is set back to “Inactive” as seen in Table 2. If no errors occur, the method returns zero.

3.4.1.17: int NcsInterface::SVC_Read ()

This method uses the service described in 3.2.6.8:. As mentioned previously it is only able to read parameters that can be converted to an unsigned integer. As an input it receives the number of the desired parameter, two pointers to integers to receive the “value” and “attribute” from the SVC parameter and the system data channel to be used. If no error occurs the method returns zero.

3.4.1.18: int NcsInterface::SVC_Read_Stress ()

This method uses the service described in 3.2.6.9:. Differently than most methods created, that work in a synchronous way with the service architecture, this method only invokes the service, it does not wait for the acknowledge flag, that has to be done manually by the user afterwards.

The necessity for this method to be asynchronous arises from the objective of a stress test which tries to read multiple times the same parameter using the NC, the PLC and also directly through the NCS interface multiple times over a period of time through the same channels.

This method receives as input the number of read operations to perform, which parameter to access, in which drive to access it and the system data channel to use. If this method accesses the system data and write the necessary inputs it returns zero. This error value does not mean the service was successful, it only means that there was no evidence of failure with the service request.

The real error parameter has to be checked manually by the user. Like the service describes, the value returned should indicate the number of access errors that happened during the execution.

3.4.1.19: int NcsInterface::NC_SVC_Read_Stress ()

This method uses the service described in 3.3.1:. It works similarly as the method described in 3.4.1.18: but is performed in the NC instead of the PLC. As explained in the service due to the limitations of the CPL language, mostly regarding the operation of strings, it was decided to fix the operation to parameter S-0-0380 to perform the stress accesses. It is also asynchronous and the error works exactly as the one described for the PLC service.

3.4.1.20: int NcsInterface::SVC_Read_List ()

This method uses NCS functions directly and it is implemented for extremely big parameter lists. That was necessary because usually small lists can be read as strings using the NCS directly, but the function used to read strings has a memory

limit defined by the MTX and for big lists the system creates a file in the NC file system containing all the information.

The method, creates the necessary files, performs a read operation of the resulting file that contains the whole list, it copies all the data to a string in the TestBuddy, which is not limited by the MTX memory and then deletes the files.

This function takes as an input, the drive id to identify from which drive the parameter list will be read; the list identification number; a pointer to an array of chars and its size. The function will write the list in the provided char array putting the "\0" character in the end of the array and if no errors happen it will return zero.

3.4.1.21: int NcsInterface::GetRelease ()

When errors happen, it is very important to provide useful information for debugging purposes. One of the most desired information the system tests can provide for debugging is the firmware version in the equipment being tested. This method access that using a special NCS function, it treats the returned data to a string and prints it in the test log. It doesn't have any input parameters and if no error occurs it returns zero.

3.4.1.22: int NcsInterface::checkDev ()

This method was created to double check and to provide a more specific diagnostic of the flag returned by the check topology method described in 3.4.1.2: and also in the service described in 3.2.6.3:. These functions explained before would provide an output Boolean with the name "AllDevs", which as the name indicates, is true when all devices in the topology are active but if one device is inactive it becomes false.

The function block used by the PLC to actually check that, is dependent on a table in its memory containing that information, and depending on some specific conditions, it is possible for that value to be wrong due to when and how it is updated. For that reason a function was created to not only identify if there are inactive devices but to also confirm which devices they are. That is done using NCS

functions to access some specific SVC parameters and comparing reserved bit values from them to the expected values for those cases.

This function receives as input an array that must be as long as the number of devices in the topology, and also the number of devices in the topology. It verifies every device and in the positions in the array, which represent the specific device, the value one (1) is written. In example if the topology has four devices and only device 2 is inactive, it will return the following array: [0 0 1 0]. The array, as in C++ starts in the position zero, which is also the same starting reference number of the Sercos topology. If no errors are detected during the function execution, it returns zero.

3.4.1.23: int NcsInterface::CheckTarId ()

In order to make the tests automatic and to ensure the functionality of the programs, every test has an “init” phase where it loads the right configuration files and restarts the system. These operations are executed to guarantee known initial conditions to perform the tests. During these phases, the upload operation for the configuration takes a relatively long time comparing it with the execution time of the test cases in itself. Taking that into account and also the fact that the initial conditions are usually the same for a group of tests and not for each of them specifically, it became interesting to be able to verify which file was loaded and if the one used for the test was already loaded, only the restart operation was needed, more than halving the time of the “init” phase.

Since the system does not provide a checksum or other verifying mechanisms for the configurations, after some discussions, it was decided to use the system data to perform that control. A System Data variable would be declared in the same declaration file used for the channels, and in each different configuration, it would be initialized with a different value. The initialization would be performed through an initialization file as the one mentioned in 3.1:. This way, since the file system is one of the things extracted from the configuration files, for each different configuration the initialization file would be different and when the system is restarted it would initialize the variable with the right parameter.

This function does not use any service but it uses System Data access to read the value of the specified string. As an input it takes a string to compare with the one in the system data, if they are equal it returns zero.

3.4.1.24: int NcsInterface::ReadWera ()

During operation, the MTX can store errors and their diagnosis in a list inside the system. Those errors are called Wera Errors and to read them, special NCS functions have to be used. This method was implemented for that reason, as input it receives a pointer to an array of structs of the type “Ncs_WeaSysInfoEvL_T” which is already defined in the NCS headers and contains all the necessary variables to store the information returned from the read function. It also receives the length of the array, and a pointer to an integer variable where it can write the number of values that were truly read.

The function needed these inputs because the read operation needs the error list to be open, similar to the way a “read line” operation needs a file to be open in C/C++. In each read operation it is as if one line of the file was read and as happens to a file after reading all the lines, in this function when all the errors are read, the pointer to the error list becomes “NULL”.

This way, since there is no way to know how many errors would be in the list, this function reads all of them, but it only writes in the array provided as input the allowed number, that is also the reason why it writes in the other variable provided the number of values that were actually in the list. This way the user can compare the difference in the array size and in the number of read values to know if the entire list could be read. If no errors are encountered, the function returns zero.

3.4.1.25: int NcsInterface::Nc_Test_Mux ()

This method was designed with a specific test in mind, the Mux test, because it was implemented originally using spaces in the MTX’s memory that are not accessible through the system data. For that reason the NC program used in the test was modified to use system data variables instead. And this method uses a service in the NC, described in 3.3.2:, to access those variables and fetch their values to be

processed in the test. As an input it takes two integer pointers to write the values of the accessed variables and also the channel to be used in the system data. If no error is detected it returns zero.

3.4.1.26: int NcsInterface::ScanDvcList ()

This method was created in order to identify how many devices are recognized in the topology. Using an internal list from the MTX which is automatically updated by the system depending on special conditions, this method verifies how many devices there are in the topology. It does not use services but access the system data directly. As inputs it takes a pointer to an unsigned short array, a pointer to an integer, and an integer which represents the start-up mode. In the unsigned short array the function will write the Ids of the devices, in the integer it writes how many they are. If no error happens the function returns zero.

This function can be used in two different modes, in “Normal” mode it will return only the configured devices, but if invoked in “Topology” mode, it will return also any unknown devices that are in the topology. By invoking it both ways and comparing the results the user can determine if all devices in the topology are configured correctly or not.

3.4.2: Utilities Class

As mentioned before, the connection with the MTX and the TestBuddy was managed in the NcsInterface by the constructor and destructor of the class, in order to simplify the hurdles of the project. The functions would take information from the system like the window definer and other information that were not trivial. For that reason, it was impossible to manage connection freely and to do so this new class was created.

It possess a NcsInterface instance as a public element that can be initialized and destroyed as many times as needed inside the class, this way making it able to connect and disconnect at will. Using that, three methods were created in the Utilities Class explained in the list below.

It is important to point out, that by the end of the project, and taking into account all the experience and knowledge gained about the systems, it is thought that the implementation could be improved by merging the classes NcsInterface and Utilities using inheritance and changing a little bit the usage of the NCS functions for connection. That way it is believed that the solution would be more elegant, however there was no time to implement it.

3.4.2.1: int Utilities::Init ()

This method is used from a situation at which the framework is not yet connected to the MTX, it establishes connection, checks the *.tar configuration file, restarts the system and keeps polling for reconnection until it succeeds or the test timeout is triggered. As input it takes a WCHAR string with address and port of the MTX divided by a ":" character, i.e. "192.168.1.203:10099" where the IP address of the target MTX is "192.168.1.203" and the port is "10099". It also receives a string with the complete path for the *.tar file, another string with only the *.tar file name without the extension i.e. if the file is called "Configuration001.tar" the user should input "Configuration001". This was done as a convention to manage the system data comparison for the configurations, the initialized value in the system data that identifies each configuration file should be the name of that file. Lastly the method takes which system data channel to use for that check. If no error is detected, the method returns zero.

3.4.2.2: int Utilities::Restart ()

This method is only an operation that reconnects to the system after the restart is done. It is very similar to the previous method with the difference that it doesn't check for the *.tar configuration file and the system must already be connected to perform it. The inputs are simply a string with the IP address and Port divided by a ":" just as in the example presented in 3.4.2.1: and the second is a system data channel to be used in order to initialize the NcsInterface's internal variables for the "Types" in the system data, if no error is detected the method returns zero.

3.4.2.3: int Utilities::Connect ()

The last method of the class is the simplest of the three; it is only used to establish connection. It does not restart or perform any other operation with the system. As an input it receives the same inputs as the Restart method presented in 3.4.2.2: and performs the connection. If no error is detected the method returns zero.

3.4.3: Maps Header

One of the objectives of the project, as mentioned in Chapter 1:, was to make the development and implementation of test cases as simple as possible. After reading Chapter 2: and Chapter 3: up to this point, the reader should be able to notice that there are many variables, Ids and channels that need to be used with specific values to request services or use helper methods and son on.

To help keeping track of all that and make it easier for the user to access these values, a header with definitions of all the values needed for the project was created with the name “maps.h”, it was called this way because the objective is for it to behave like a reference map for all the needed functionalities, i.e. the service Id’s are defined in an enumeration that must be updated to be equal to the one in the PLC with meaningful names. Another example is a Channel type enumeration, that defines the declared channels in the System Data, and if used in the same way as in the methods described previously, as an input to the functions, it can prevent access to undeclared Channels before compiling the code, since only the existing Channels should be present in this enumeration. The header also contains the same bit map presented in the PLC, in order to simplify the request of services.

Moreover, definitions that are used in many tests, i.e. MTX string size or array size for SVC access, wera errors reading, and other functionalities are also defined in this header. The header can be found in appendix B for those interested in the specific definitions and declarations used in the project.

Chapter 4: Test Cases

As mentioned in Chapter 1:, implementing test cases that were previously performed manually was one of the main objectives of the project. Before explaining each test in more detail, it is necessary to present an overview of how tests are structured and treated in TestBuddy.

In the framework, every test is treated as an individual class with three standard methods: doInt, doTest and doEnd. These methods are automatically executed in this order by every test as the default operation of the TestBuddy. They were meant to divide the operations and simplify the development of the tests.

Since they are actual classes, in every test it is possible to declare public and private variables or objects, and in this project every test that was implemented possesses a pointer to a not initialized instance of the “Utilities()” class, as a private object. This way, in every test any of the three methods have access to this object and can initialize or delete it depending on the need, usually the instance is initialized in the doInit method and destroyed in the doEnd method to free the memory.

The framework already possesses a “test class” that provides some macros to operate functions from the TestBuddy like printing logs, errors and other useful operations as well as some macros that are necessary to “register” the tests in the framework. To put it simply, in Figure 20, shown below, there is an example of the framework window with the test list, if the test is not registered correctly, it won’t be shown in the list, independently of the source files being in the VS project or not.

The idea for the tests that were implemented in this project, was to follow the structure already defined, using the doInit method to put the system in a desired initial state, then in the doTest method execute the operation and finally in the doEnd reset or delete any variables or conditions that were altered. In this respect, the doInit method became almost standard, where the test would connect to the MTX, check if the correct configuration file is loaded, if yes it would restart the system, if not, before restarting, it would perform the upload of the correct configuration.

Usually for most tests performed in the project, the initialization was the most time consuming operation.

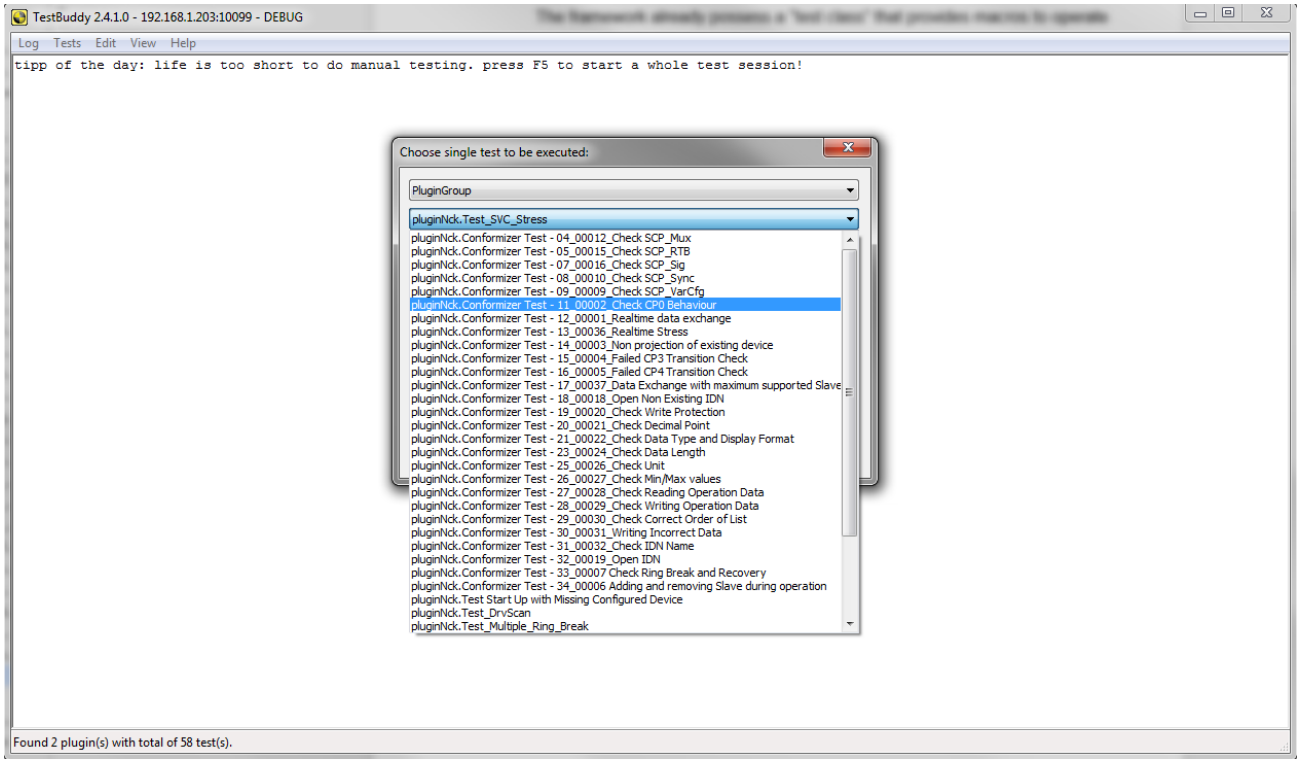

Figure 20 - TestBuddy Window With Test List Open

Most of the implemented tests in the project were taken from a list of 34 test cases that were already performed manually, not all of the list could be implemented due to restrictions on the test equipment described in 1.4:. The rest of the implemented tests were suggested for situations found interesting during development of the project. Below, each test will be briefly described.

4.1: Tests from the List

This section is dedicated to the tests taken from the list of tests that can be seen in appendix C, the document is in its original version.

4.1.1: Conformizer Test – 04_00012 Check SCP Mux

This test was meant to check if multiplexing data to and from a slave is possible. It is implemented by setting specific SVC configuration parameters in the MTX and then running a NC program that performs the necessary operation and store values in two system data variables that can be accessed by the test using the service described in 3.3.2:.

Since it has to load SVC configurations that are different than the default values, in the beginning of the test, it fetches the original values, store them in variables that will not be tampered with during the test. Then the test writes the new configuration parameters in the system and executes a system restart in order to make them active. After these operations are done, it loads the correct NC program and executes it. After the program has finished, it takes the two inputs from the system data and compare to the expected values.

This test uses methods to load, start and wait the NC operation channel described in section 3.4.1:. To perform the SVC access and modification it uses the NCS libraries directly.

4.1.2: Conformizer Test – 05_00015 Check SCP RTB

This test verifies if real time data bits can be configured between master and slave. For this test, two SVC parameters need to be modified, and in total 6 parameters are verified and compared to expected values. Aside from the two that need to be modified before execution, the other four parameters should be set automatically.

The test fetches the two parameters that will be modified and store in variables that won't be changed, then it applies the modifications and restarts the system. Afterwards, the six variables are fetched and compared. Once the test is over, the initial values from the modified variables are returned to the system.

4.1.3: Conformizer Test – 07_00016 Check SCP Sig

This test checks if it is possible to configure a signal control word correctly. It changes four SVC parameter configurations. As the previous tests it stores the initial configurations in the beginning of the test to restore them afterwards, then it writes the desired configurations in the system and read them again to ensure that the written values were accepted.

4.1.4: Conformizer Tests – 08_00010 Check SCP Sync Configuration; 09_00009 Check SCP VarCfg Configuration; 12_00001 Real Time Data Exchange; 13_00036 Realtime Data Stress Test.

These four test cases are performed using the same procedure to check:

- If the master configures parameters for SCP_Sync correctly and real time data can be exchanged.
- Configures connections with a SCP_VarCfg slave and check if real time data exchange to and from the slave is working correctly by setting and writing values in CP4 (Operation Phase).
- Check if master can exchange real time data with one device.
- Configures a heavy amount of data and then check if the master can still exchange data.

The tests procedure consists of running a program that moves two drives, from axis X and Y, from position 0 to 100. During the operation, the test accesses the system and verifies the positions stored in the drives in real time and copy them to arrays. Once the NC program finishes its operation, the arrays are checked to see if the positions respect the desired behaviour. Moreover, the speed of the drives is estimated and compared to the reference speed used in the program to ensure that the drives were storing valid positions.

These tests need to actually perform axis motion with the drives, so it must use the services described for turning power on, setting the drives ready and also the methods to load and run the correct NC program.

Since these tests would perform the same operation, but had to be created separately for administrative reasons, a helper class specific for them was created with “doInit”, “doTest” and “doEnd” methods that could be invoked in all the test cases easily, without repeating code. That class was called “Conformizer_8_9_12_13”.

4.1.5: Conformizer Test – 11_00002 Check CP0 Behaviour

This test checks if the master detects devices correctly in CP0. In order to perform this test, the method described in 3.4.1.26: is used. The test issues a phase change for CP0 and then executes the service in Normal mode, then it goes to phase two in Topology mode and verifies the device list again. With both results, it performs a comparison between them, and if both are equal, the test succeeds. If they are not, the sercos ids from the unrecognized devices is returned in an error log.

4.1.6: Conformizer Test – 14_00003 Non Projection of existing device.

This test checks if an error is produced during phase change when a slave that is not projected, which means to say a device that is not configured in the IWE program that is running in the MTX, is connected to the topology. To do it, this test needs a configuration file different than the standard used in most tests. In this configuration, one of the drives in the test rack described in 1.4: is eliminated from the IWE configuration, ensuring it will be an unidentified device for the master.

Once the correct configuration for the test is loaded, the system is restarted and the test verifies the expected behaviour through Wera errors reading. For this situation, two specific errors should be triggered in the start-up of the system.

4.1.7: Conformizer Test – 15_00004 Failed CP3 Transition Check

This test verifies if the master is able to detect a failed procedure command “CP3 transition check” and stays in CP2 cancelling the normal start up procedure that goes until CP4.

In order to perform this test, specific SVC configurations with wrong values are loaded to the controller and then the system is restarted. In order to set all the default configurations back after the test, as in the cases described before, the initial conditions are stored in the beginning of the test.

Once the system is restarting with the wrong configurations, the test verifies if the right diagnose was performed by the MTX by checking if the expected error is set in the system.

4.1.8: Conformizer Test – 16_00005 Failed CP4 Transition Check

This test checks if the master detects the failed procedure command “CP4 transition check” and stays in CP3. It has the same structure as the previous one, but it modifies different parameters to force other expected errors in the system, this time to stop the start-up procedure between CP3 and CP4.

4.1.9: Conformizer Test – 17_00017 Data Exchange with maximum supported slaves over SVC.

This test checks if the master can exchange data with the maximum number of supported slaves. It is done by randomly choosing 4 of the devices in the topology and accessing a specific parameter from them to verify if the read operation succeeds. The idea is that randomly choosing four devices would be enough, but now that the test is automatic the logic could be easily changed to simply scan all devices, which would make it more reliable.

4.1.10: Conformizer Test – 18_00018 Open non existing IDN

This test tries to access a parameter that does not exist and verifies if the correct error is sent back. It is done directly through NCS and there is no need to change configurations. It is one of the fastest tests that were implemented because of its simplicity.

4.1.11: Conformizer Test – 19_00020 Check Write Protection

This test, tries to access and write a parameter that is write protected in different phases, more specifically in CP2, CP3 and CP4. Normally write operations can only be performed in CP2, so every parameter is already write protected in CP3 and CP4, but for this test a parameter that is also protected in CP2 is used to ensure that the write protection from the system works correctly in any case.

4.1.12: Conformizer Test – 20_00021 Check Decimal Point

This test checks a SVC parameter that store values with floating point numbers and evaluate if they contain the right format and if the decimal point is in the right place.

To do it, the test only uses methods from 3.4.2 and direct access with NCS but the original test would also need to operate the information of the system through a second, redundant, channel, which uses a hardware connection through what is called as an “Engineering port”. This type of connection wasn’t available for the project, therefore it was left for the team to decide in the future if there is a need to upgrade the test case or if it is good as is.

4.1.13: Conformizer Test – 21_00022 Check Data Type and Display Format

This test reads parameters with different data types like, hexadecimal, binary, a string with UTF-8 format, an IDN list and some numeric types like unsigned values, signed values, and floating point numbers. It uses Regex expressions to verify some of the formats, and specific numeric or string comparisons to validate the rest.

In this test, to ensure that the system possesses a string in UTF-8 format, a string with special characters is written to the system in the beginning of the test. Because of this reason, the test also has to store its initial value to be reset afterwards.

4.1.14: Conformizer Test – 23_00024 Check Data Length

Check if the data length is displayed correctly and that only parameters within the data length are accepted through write operations. This test operates with six SVC parameters that need to be written to and therefore need to be stored for reset after the test. It tries different types of inputs with the parameters as well as correct and incorrect values, and for each situation, it checks the expected behaviour.

4.1.15: Conformizer Test – 25_00026 Check Unit

Every SVC parameter is referred to as an IDN, and each of them possess some other descriptor fields aside from the value, for instance the field “Unit”, which stores information like “mm” or “%” and so on, used to give meaning to the value parameter. This test accesses three parameters with different unit types and compares their units with the expected ones.

This test also uses only the SVC access through NCS while the original version would also use an Engineering port and was marked for further evaluation in the future.

4.1.16: Conformizer Test – 26_00027 Check Min/Max Values

As mentioned for the previous test, there are other descriptor fields aside from the value of the IDNs. This test verifies two parameters for three different IDNs, the minimum and maximum value that IDN accepts. These two parameters work as the limits of these IDNs.

This test, originally, would also double check that information with the engineering port and was marked for further evaluation of this double check operation.

4.1.17: Conformizer Test – 27_00028 Check Reading Operation Data

For this test, a set of parameters should be read to ensure that the operations were performed correctly with operation data. Therefore the test performs a set of

read operations in six different IDNs and only analyses if the operations were successful, without comparing the read values.

Originally, the same values would be read and cross checked with the ones from the engineering port and because of that this test was also marked down for further evaluation.

4.1.18: Conformizer Test – 28_00029 Check Writing Operation Data

This test is similar to the previous one, but instead of verifying the read operations it verifies the write ones. Originally it would write through the SVC parameters interface in the master, which is accessed by the NCS and then the value would be verified using an engineering port. Since that hardware wasn't available as mentioned before, the test was implemented using the NCS access to write and to check if the right values were written. But since it is different than the original version, this test was also marked for future evaluation.

4.1.19: Conformizer Test – 29_00030 Check Correct Order of List

This test reads an ordered list that contains all the IDNs available from the system and checks if it respects the rules of ordering. The list is contained in a specific IDN which can't be read directly by the NCS as in the tests previously mentioned because it is too big. It uses a method described in 3.4.1.20: devised specifically for these situations where the size of the exchanged data was too big. The service copies the whole list to a string that is afterwards analysed.

4.1.20: Conformizer Test – 30_00031 Writing incorrect Data

This test is very similar to test 23_00024 in the respect that it tries to write wrong data to the system. In this case however, the data would have the correct size but with unaccepted values. After the write operations, the values should be checked through the engineering port, but for this test, again, the NCS access was used for both purposes and for this reason the test was also marked for further evaluation. In this test it is also necessary to save the initial values of the parameters in the beginning of the test just in case to reset them afterwards.

4.1.21: Conformizer Test – 31_00032 Check IDN Name

This test verifies if the IDN name, which is another attribute of the IDNs just like the unit, min or max values and so on, is correctly displayed in the system. To do so, it fetches the names of three specific IDNs and compares with the expected values they should display. In the original version the engineering ports were also used to double check the values and compare with the correct ones and once again the test case was marked down to further evaluation.

4.1.22: Conformizer Test – 33_00007 Check Ring Break and Recovery

This test is not related to the parameters in the system as most of the previous ones, its objective is to verify if the system is able to switch between the correct topologies in reaction to extern events i.e. a cable break. This test uses services to check topology and control a cable break and restore operation. It checks that the system is in topology mode before starting, breaks the ring, checks if the system goes to double line topology and also that it recognized the ring break position correctly. Then it restores the ring, and verifies if the system is capable of re-establishing the ring topology, what is also referred to as “Ring Healing” in the company.

4.1.23: Conformizer Test – 34_00006 Adding and Removing Slave During Operation.

In regards to connecting the drives with the master communication device, the company provide devices with two types of operation methods, normal connection, in which the device has to be already connected with the master since the beginning of the start-up operation and has to be configured in the IWE. And it also offers devices that use what is referred to as “Hotplug” connection, which are devices that can be connected to the topology after the system has already been started.

For this test, only drives that don’t support Hotpluging should be used otherwise the test will fail. The test starts up normally with all the drives in the test rack, then it uses the service structure to turn off one of the drives, check to see if

the system recognized that change and then turns the driver on again. The expected result is that the system displays a specific error and the drive that was toggled should display an error message in its screen, but the other drives should continue to be operational.

4.2: Other Tests

4.2.1: Test - Multiple Ring Break.

This test is an improvement of the test described in 4.1.22:. The Idea for it came from the fact that, as a safety measure to the topology and to ensure the real time constraints of the system, the sercos device keeps track of how many times the ring was broken in a window of 10 minutes. If the ring is broken more than two times during that window and recovers (physically speaking), the Sercos device will stop healing the ring connection and leave the system as a double line. This is a measure to prevent against faulty wiring in the machine that might not have broken completely, resulting in bad contacts for instance.

This test verifies if the ring is broken correctly two times and healed, and in the third time it checks if the system respect that safety feature and refrains from healing the ring topology.

4.2.2: Test – Start Up With Missing Configured Device.

In the test list presented before in 4.1.6:, the system is started up with an unknown device and its reaction is checked. This test verifies a slightly different situation, where there is a device in the program configuration of the system, but that device is not physically present. In this situation, the test also needs a different configuration file from the standard. The expected behaviour of this test is to detect a specific error but even so put all the drives ready for operation in CP4.

4.2.3: Test – NC Program.

This test verifies if the system can run an Nc program, then stop it, restart the channel and run another program without mistakes. At least one of the programs used has operations with the axis movements so the Test needs to use the service architecture to set the test rack.

4.2.4: Test – Phase Change.

This test verifies if the system can access any of the phases from CP0 to CP4 and also tries to change the operating mode from OM to PM in all of them. In phases below CP2 the system can't access PM mode, therefore the test verifies if this restriction is respected. From CP2 onwards the expected result is for PM mode to be set normally.

4.2.5: Test – Phase Change With Missing Device.

This Test turns off one device during operation, and it verifies if all phases are still reachable by the remaining devices in the topology. It is very similar to the previous test but it does not verify OM and PM switching. The expected behaviour is for all the phases to still be reachable.

4.2.6: Test – Ring Recovery in CP2

During the development of the project, a bug was reported from another team that the MTX system was unable to heal the ring in CP2, therefore a test exactly like the one presented in 4.1.22: was developed, except that it changes the system to CP2 before starting. The same results are expected during this test and at least for the equipment configured for this project, the reported bug never occurred.

4.2.7: Test – SVC Stress

This test verifies if there are any conflicts while reading SVC parameters from different places using the same structures and accessing the same parameter. To do this, the test uses two asynchronous methods described in 3.4.1.18: and 3.4.1.19:. It

uses the IDN “S-0-0380” because of a restriction in the NC service. This IDN was chosen because it can be read as a number directly, and its behaviour was known well since it is used to verify the voltage that the drives receive in other tests that operate with the NC.

The test performs 1000 read operations through NCS, the PLC and the NC totalling 3000 accesses from systems that would work entirely in parallel. These operations are done in the same time window and happen independently with no synchronization at all, the expected behaviour is for all the systems to read 1000 times without any error or communication conflict even though they are accessing the same information.

4.2.8: Test – SVC Access With Wrong Conditions

This test forcefully puts the system in two types of conditions that are not default and expects the access to the SVC parameters to be completely impossible in one of them and in the other the system should handle the situation and still be able to perform read operations. One of the situations is during a NC channel restart, in which the parameter should still be accessed, and the other is during phase change, which should deny the access.

Chapter 5: Lua Evaluation.

Lua is a very powerful, light and fast scripting language. It was created, and is maintained, by a team at the Pontifical University of Rio de Janeiro in Brazil (PUC-Rio). This language combines simple syntax with very powerful data description constructs based on associative arrays and extensible semantics. Moreover, it is dynamically typed and runs by interpreting byte code for a register-based virtual machine. More information about the Language can be found in [2] and [7].

It was considered interesting for this project specifically for the qualities presented above. First it is fast and would not compromise the efficiency of the code. Second it is light and easily portable, it doesn't need a big number of libraries. Third, it has great structures to manage variables and data. However, lastly and above all else, it is considered a simple language to work with.

Since one of the objectives of the project was to take as much complexity away from the user as possible, the idea was to evaluate if Lua would present an easier interface than C++ for unexperienced programmers.

5.1: TestBuddy and Lua

To operate test cases using Lua, a plugin for the TestBuddy was already created by another team, as mentioned in Chapter 1:. This extension runs a virtual machine to process the scripts and execute tests. It already offers some basic operations from the TestBuddy that in C++ are provided by special classes from the framework to be used in Lua. In order to work in the scripts, these functions needed to be "wrapped" in the PluginLua. Wrapping is a technic very common when working with different languages, through it, the Lua scripts are able to invoke functions from C++ which makes it easier to use code that is already implemented. In order to create these wrapped functions, the Lua API was used, it provides all the necessary tools to enable interaction between Lua and C/C++. It is provided in the Lua official reference manual, more details about it can be found in [2].

In order to be recognized, every test has to be registered, as it was presented in Chapter 4:, but the method to perform that operation in Lua is a little different. In this case, the script has to return a table containing a minimum set of information:

- Test Name
- Description
- Responsible
- Substitute

If the script returns the necessary data, the PluginLua takes care of the rest, but without those informations the test cannot be registered.

The operation for the test is also divided in three methods named doInit, doTest and doEnd in the same way it is done in C++.

For this project, it was not enough to create, modify and operate tests in Lua. These programs should be able to communicate with the MTX, otherwise they wouldn't be useful. But the communication with the MTX was performed through the NCS library interface, which was implemented in C++, therefore in order to operate its functions, the Lua scripts would need "wrapped" functions. These functions would use the Lua API for C/C++ to implement the conversion between both languages variables and also operate the methods in C++.

Using the Lua API, the wrapper methods can dispose of a stack to send or receive information to and from Lua. It also provides functions to help with the type conversions between the languages. That stack can be accessed using as reference both the top and the bottom of the stack and allows any of its values to be accessed, so it is not necessary to clear one value from the stack to be able to reach the next. The elements it can contain normal variables like strings and numbers, and also more complex structures, like arrays, tables or even a complex structure inside another, as for instance a table containing many arrays or an array of tables.

Lastly, it is important to notice that, implementing the wrapped methods might imply in some complexity, but after they are functional, the end user can use them with relative simplicity.

5.2: Lua X C++ Comparison

As mentioned in Chapter 1:, in order to evaluate both languages, it was decided to take a test considered to have “intermediate complexity”, wrap only the necessary methods, and implement it in Lua. The chosen test was the “Multiple Ring Break Test” presented in 4.2.1:. In order to do it, five wrapper functions were created, the functions described in 3.4.2.1: “Init”, 3.4.1.3: “SetIO”, 3.4.1.2: “CheckTopology”, 3.4.1.22: “checkDev” and lastly a wrapper function was created to access a global variable from the framework, which is initialized when the TestBuddy is run, with a path to either the execution folder, or a path provided by a command line to indicate where the configuration files should be stored. This last wrapper function doesn’t invoke any methods, only accesses a variable and sends its value through the stack back to Lua.

After performing this test, in both languages, a comparison was done regarding the obstacles set in both of them. To simplify their explanation, the advantages and disadvantages will be presented in the format of a list.

5.2.1: C++ Characteristics

5.2.1.1: Advantages:

- Offers all the flexibility supported by the language without needing any special configuration.

  There is no need to use or create any additional feature like wrapper methods.

- Creating new functions or functionalities is relatively straight forward.

  Inside the IDE to implement new features it is simply a matter of adding structures, classes, including libraries directly and so on.

- Debugging is supported by the VS Pro directly.

  With the right configurations set, the VS Pro supports debugging directly.

5.2.1.2: Disadvantages:

- Need Visual Studio Pro to compile changes.

  For now, the framework is based on libraries that are only available in the Visual Studio Pro IDE, the Express version for instance is not enough. Moreover, there are some changes in the operation of the VS depending on the version, for instance, the testbuddy doesn't compile in VS 2015 due to the removal of some properties and libraries. This makes developing the framework an activity not only coupled with visual studio pro, but also with the 2012 release of the program.

- Every change done is only recognized after compiling.

  Every time a change is done in the code it is mandatory to re-compile for it to be recognized. This isn't a really big drawback because with the current processing power of most computers it is done pretty fast.

- Can represent an obstacle if the users are not familiar with the language.

  C/C++ are not considered really high level languages especially concerning variable and type manipulation. Functions with multiple outputs need to work with the concept of pointers and the conversion between types can be non-trivial depending on the situation. If the objective is to be simple for non-expert users these concepts may prove challenging.

5.2.2: Lua Characteristics

5.2.2.1: Advantages:

- Tests can be altered without re-compiling.

  Once the framework is compiled, Lua tests can be added, changed or deleted by adding or removing them from the specified directory that will be used by the executable file to search for the *.lua archives.

- Doesn’t need a specific IDE

  The fact that it doesn’t need to recompile the project to implement tests frees the Lua implementation of the need to work with visual studio professional. This might be interesting depending on the costs related to licensing with the IDE.

- Manipulation of types and variables is done in a higher level. It becomes rather simple.

  This language is typed dynamically, therefore the need to define sizes for variables or arrays is taken from the user and is done automatically. Not only that, but variables in Lua don’t need to be declared, they are automatically declared upon initialization and the type handling is also autonomous. Lua provides a simple but powerful base with really simple data structures that can be used to treat data in the most varied of sorts and its functions can return any type of outputs as well as multiple outputs without using the concept of pointers.

  The functions can return more meaningful information for errors and other situations in simpler ways. Even functions can be returned as outputs from other functions. Also, any name is modifiable therefore a function can be modified, created or deleted during execution.

5.2.2.2: Disadvantages:

- Debugging without an IDE is extremely difficult.

  It is not ideal that the IDE that needs to be used to work with C++ is restricted, but the freedom in Lua comes with a price. It is not integrated directly with the framework, therefore, the tests are run by the testbuddy when the program is opened but there is no way to use breakpoints in the code or other debugging options at this point.

  The wrapped functions are also not visible when programing the Lua scripts because they are defined in the Visual Studio and the scripts

are only taking them after being compiled, therefore IDE helper options as word completion or method search can't be used.

Also, since there are no debugging options, there are some IDEs like Eclipse for Lua that can be used for syntax checking, but aside from that, every other debugging operation must be done manually. This is less than ideal for inexperienced users.

- Wrapping

  As good a tool as it is to interact with other languages, wrapping is still something that would need to be done to every function from the framework libraries or the service architecture that needs to be used. This would not only demand a lot of work, depending on the necessities, but it would also require the Visual Studio Pro.

  Unless every function needed by the Lua implementation was already wrapped, the programmer would still need Visual Studio Pro.

- Some operations with the TestBuddy seem to present erratic behavior.

  Since the PluginLua was implemented rather recently, and not many applications use it already, it might present bugs or flaws that were not encountered yet. As for example, in this project, if a Lua script is run in the framework when it is just opened, printing operations to the test log work fine if functions in Lua are used, but if those printing operations are performed from inside a wrapped function, they are not recognized. However, if a Test that runs in C++ is run, and perform a print operation, then the prints from inside the Lua wrapped functions begin to work.

5.3: Conclusion

Using Lua proved really interesting regarding types and variables manipulation. It has no need for pointers, the functions can return any kind of data supported by the language and also in any quantity. Using this and defining a standard method for the output. i.e. Defining that all functions created for the project

should return a table with at least some specific parameters like “Error”, “Error_Message” and so on, could make using those functions a lot simpler, instead of every function having its own set of pointers, with different types. Another great simplifier for the creation of tests and structures is not having to define sizes for variables, arrays and so on, since they are dynamically created, making it more versatile.

It is also not necessary to recompile the project in order to create tests and work with them. This would be good to reduce costs with licenses. Assuming that the Lua implementation was in a stage that new functions would be needed rarely and the users would be able to develop new functionalities with the provided functions.

Unfortunately, at this point, the pluginLua does not provide integration with any IDE to enable good debugging options. Moreover, the plugin presents erratic behaviour, making its usage less interesting, especially concerning non expert users when it comes to manually debugging.

An example of unexpected functioning that was found and has a really bad impact in the decision to use Lua, is that TestBuddy doesn’t recognize the pluginLua automatically in any computer. If the solution is cloned from the Git, with the exact same procedure, it works fine in some machines, but can present a bug in which the plugin is not found, impacting the portability of the solution.

Therefore, after the analysis, the conclusion is that even though the usage of the language could be interesting regarding simplicity for new users, the fact that no IDE can be used to provide good debugging options, associated also with the fact that the portability and other erratic behaviours were identified, puts the language in a position where it is not worth using, at least in the current situation.

Chapter 6: Results and Conclusion.

In the end of the project, the planned development was accomplished. The service architecture was created and is functional, disposing of 12 services. Many automated tests were implemented, from a list of 34 tests, 23 were fully implemented and 8 new tests were suggested and are completely operational.

However, because of limitations on the equipment used for the project, the remaining tests from the list could not be fully implemented or needed to be adapted and will need to be revised in the future.

Moreover, the evaluation of Lua as an alternative programming language for the tests was performed and presented a negative result for the project with the current conditions. If the usage of better debugging tools is made possible in the future, and if the Lua plugin's behaviour becomes more stable, it is suggested to re access the viability of the language according to the benefits presented in Chapter 5:.

The advantages presented by the implementation of these automated tests are in theory the ability to repeat the same tests with standard quality as many times as needed. Saving time from the employees, since they can just start a sequence of tests and leave the program executing while doing other activities. In this respect, it is also possible to use them as nightly tests, to keep running during the night and perform stress tests of the systems, maybe repeating the same test over and over to verify if the results are consistent.

The tests implemented in this project have an average execution time below 4 minutes, with a few exceptions. Every test performs a system restart before executing to ensure that all the variables and flags are in the correct state, this couldn't be avoided and represents around 60% of the time spent for most tests, which is around 2 minutes.

The tests are able to load the correct configuration files to the system before restarting, which is a very important ability in order to automate them. This operation usually takes the same or more time than the restart itself, but a solution was

devised to check if the correct files are in the system, in which case, this operation is skipped, saving about two to three minutes per test.

Moreover, now that the service architecture is functioning, in order to implement new tests that use functionalities from the PLC or NC components from the MTX should be easier and simpler, therefore contributing for the future developments of the department regarding auto tests for sercos. It is also a good result regarding the department operation using test driven methodology.

On the other hand, it is noted that an obstacle to the usage of the TestBuddy framework to perform tests for the MTX systems is the lack of documentation of the NCS libraries and the difficulty to find already implemented functions to be used in order to provide useful information for the tests. This was by far the greatest obstacle in the project.

Finally, after finishing the project and analysing its implementation with all the experience and knowledge acquired, there are some suggestions of changes that would make the whole implementation cleaner and more efficient. For example to change the connection methods and unify the NcsInterface Class and the Utilities Class making the solution cleaner. There are other operations that could be improved by changing the equipment in the Test rack and integrating it with the test structures.

Overall, it is believed that the project was successful and achieved the desired results. Most of the objectives were achieved and the system as a whole is working. From this point onwards it is thought that the structure can be further developed by the team to create new test cases and provide even better systems.

Bibliography:

[1] “IndraMotion MTX The CNC system solution for perfect cutting and forming”, Bosch Rexroth AG, Lohr am Main, Germany, 2012.

[2] R. Lerusalimschy, L. H. de Figueiredo, W. Celes, “Lua 5.3 Reference Manual”, Lua.org, PUC-Rio, 2015-2016.

[3] “IndraWorks 14VRS – WinStudio 7.3+SP4”, Bosch Rexroth AG, Lohr am Main, Germany, 2015.

[4] “IndraMotion MTX 14VRS Functional Description – Special Functions”, 3rd edition, Bosch Rexroth AG, Lohr am Main, Germany, 2015.

[5] “Rexroth IndraMotion MTX 13VRS PLC Interface”, 2nd edition, Bosch Rexroth AG, Lohr am Main, Germany, 2013.

[6] “Rexroth IndraMotion MTX micro 13VRS Standard NC Operation”, 2nd edition, Bosch Rexroth AG, Lohr am Main, Germany, 2014.

[7] Lua, “”The Programming Language Lua”, Lua.org, [Online]. Accessed: 28 Jul 2016.

Appendixes

Appendix A – XML Files

1) Definition File

```
<?xml version="1.0"?>
<!-- edited with XMLSPY v2004 rel. 2 U (http://www.xmlspy.com) by Markus Wehner (Rexroth Indramat GmbH) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
	<xs:include schemaLocation="basic_ty.xsd"/>
	<xs:complexType name="CN_t">
		<xs:sequence>
			<xs:element name="SvId" type="Int_t">
				<xs:annotation>
					<xs:documentation>Channel 1 service Id</xs:documentation>
				</xs:annotation>
			</xs:element>
			<xs:element name="Error" type="Int_t">
				<xs:annotation>
					<xs:documentation>Channel 1 service error status and code</xs:documentation>
				</xs:annotation>
			</xs:element>
			<xs:element name="DataIn">
				<xs:complexType>
					<xs:sequence>
```

<xs:element name="Data" type="Int_t" minOccurs="0" maxOccurs="10">
<xs:annotation>
<xs:documentation>Channel 1 input Array</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="DataOut">
<xs:complexType>
<xs:sequence>
<xs:element name="Data" type="Int_t" minOccurs="0" maxOccurs="10">
<xs:annotation>
<xs:documentation>Channel 1 output Array</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="ExecStatus" type="Boolean_t">
<xs:annotation>
<xs:documentation>Used to indicate a service request in progress</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
</wsdl:types>
<wsdl:message name="ExecRequest">
<wsdl:part name="parameters" element="tns:ExecRequest" />
</wsdl:message>
<wsdl:message name="ExecResponse">
<wsdl:part name="parameters" element="tns:ExecResponse" />
</wsdl:message>
<wsdl:portType name="ExecPortType">
<wsdl:operation name="Exec">
<wsdl:input message="tns:ExecRequest" />
<wsdl:output message="tns:ExecResponse" />
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="ExecBinding" type="tns:ExecPortType">
<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="Exec">
<soap:operation soapAction="Exec" />
<wsdl:input>
<soap:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="ExecService">
<wsdl:port name="ExecPort" binding="tns:ExecBinding">
<soap:address location="http://localhost:8080/ExecService" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```
```

```
                    </xs:annotation>
                </xs:element>
                <xs:element name="AckStatus" type="Boolean_t">
                    <xs:annotation>
                        <xs:documentation>Used to indicate that a service request has been processed</xs:documentation>
                    </xs:annotation>
                </xs:element>
            </xs:sequence>
        </xs:complexType>

        <xs:complexType name="TarStr_t">
            <xs:sequence>
                <xs:element name="TarStr">
                    <xs:simpleType>
                        <xs:restriction base="isoLatin1String">
                            <xs:annotation>
                                <xs:documentation>STRING FOR TarFile</xs:documentation>
                            </xs:annotation>
                            <xs:minLength value="0"/>
                            <xs:maxLength value="150"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
            </xs:sequence>
```

```
        </xs:complexType>
</xs:schema>
```

2) Declaration File

```
<?xml version="1.0" encoding="UTF-8"?>
<SDDef>
        <Variable Storage="volatile">
                <Name>CN1</Name>
                <Type>CN_t</Type>
                <Comment>Instantiating a channel structure to communicate using the system data</Comment>
        </Variable>
        <Variable Storage="volatile">
                <Name>CN2</Name>
                <Type>CN_t</Type>
                <Comment>Instantiating a channel structure to communicate using the system data</Comment>
        </Variable>
    <Variable Storage="volatile">
        <Name>TarIdent</Name>
            <Type>TarStr_t</Type>
        </Variable>
</SDDef>
```

3) Initialization File

```
<?xml version="1.0" encoding="UTF-8"?>
<SD>
    <TarIdent>
      <TarStr>Tar001</TarStr>
    </TarIdent>
</SD>
```

Appendix B – Header “maps.h”

```
#ifndef _maps_h

#define _maps_h


//Enum to map the bits for IOs available in the PLC to redable names.

typedef enum

{

	RING_BREAK_1=1,

	RING_BREAK_2,

	TURN_OFF_DRIVE

}BitMap;


//Enum to map the available Services in the PLC and NC with readable names

typedef enum

{

	SERVICE_IO = 1,

	SERVICE_DRIVE_POWER_ON,

	SERVICE_CHECK_TOPOLOGY,

	SERVICE_SET_OVERRIDE,

	SERVICE_DRIVE_READY,

	SERVICE_NC_STOP,

	SERVICE_NC_START,

	SERVICE_SVC_READ,

	SERVICE_NC_SVC_READ,
```

```
	SERVICE_SVC_STRESS,
	SERVICE_SVC_LIST,
	SERVICE_TEST_MUX
}ServiceMap;

//Enum to define and restrict the channels that can be used to request services from the PLC and NC
typedef enum
{
	CN1 = 1,
	CN2
}Channel;

#define SD_IF_DATA_SIZE		10		//Define the size of the array that the service architecture is able to use as input and output, this is defined in the XML schema files,
//so before changing it here, it is necessary to cross check the information.
#define Ndev 4				//Define the number of drives in the test reck

#define SD_SYSSCSDEVICE_LEN (165)
#define SVC_PARAM_MAX_LEN (100)
#define WERA_LIST_LENGTH (10)

#endif // #ifndef _maps_h
```

Appendix C - Test cases

SCP Ext Mux/ctm/00013

Test* - *1_00013_Check SCP_Ext_Mux

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00013
Link to specification	https://wiki.sercos-service.org/current/SCP_Ext_Mux
Full name	Check SCP_Ext_Mux
Description	Check if mutiplexing data to and from a slave is possible. To do this configure two connections (one for receiving and one for transmitting) with a device from the testing base and use these connection to multiplex data to an from the slave. The used container will be protocolled.
sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_Ext_Mux supported
Attributes	
Preconditions	

Actions	**A1** Connect a slave from the testing base, which supports SCP_Ext_Mux, to the master (and has an engineering port to display parameter during operation). **A2** Change to CP2. **A3** Configure two connections (one connection for multiplexing two data container to the slave and one connection for multiplexing two data container from the slave). **A4** Select atleast two parameter from S-0-0445 IDN-list of configurable data in the MDT data container (if parameter is not supported use S-0-0188 IDN-list of configurable data as consumer instead) and atleast two parameter from S-0-0444 IDN-list of configurable data in the AT data container (if parameter is not supported use S-0-0187 IDN-list of configurable data as producer instead) and configure these parameter for multiplexing to both data container. **A4** Change to CP4 and activate connections. **A5** Multiplex first parameter to the slave through first data container. **A6** Multiplex second parameter to the slave through second data container. **A7** Display first multiplexed parameter from the slave through first data container. **A8** Display second multiplexed parameter from the slave through second data container. **A9** Protocol used container and parameter.
Expected results	**E5** The slave should react according to / display over engineering port the multiplexed data. **E6** The slave should react according to / display over engineering port the multiplexed data. **E7** First multiplexed parameter should be displayed correct (be equal to information shown over engineering port). **E8** Second multiplexed parameter should be displayed correct (be equal to information shown over engineering port).
Teardown actions	none

Workflow: Finished

SCP FixCFG/ctm/00008

***Test* - 2_00008_Check SCP_FixCfg configuration**

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00008
Link to specification	https://wiki.sercos-service.org/current/SCP_FixCFG
Full name	Check SCP_FixCfg configuration
Description	Configure two connections with a SCP_FixCfg slave (one connection for receiving data and one connection for transmitting data) from the testing base. Then check if realtime data exchange to and from the slave is working correct by setting and reading values in CP4.
sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_FixCfg supported
Attributes	
Preconditions	

Actions	**A1** Connect a slave from testing base, which supports SCP_FixCFG and has connected outputs with inputs (e.g. I/O device), to the master. **A2** Change to CP2. **A3** Configure two connections (one connection for setting the outputs and one connection for receiving the inputs). **A4** Change to CP4 and activate connections. **A5** Set outputs over realtime data. **A6** Read inputs over realtime data.
Expected results	**E5** Outputs should be set. **E6** Inputs should be equal outputs.
Teardown actions	none
Workflow: Finished	

SCP HP/ctm/00011

Test - 3_00011_Check SCP_HP

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00011
Link to specification	https://wiki.sercos-service.org/current/SCP_HP
Full name	Check SCP_HP
Description	Check if one slave can be hotplugged by the master during operation. A slave from the testing base will be added at the end of the line. The master then needs to hotplug the slave correctly in CP4. Do this depending on the possibilities of the master in single line and double line.

sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_HP supported
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which supports SCP_HP and can display the actual phase, in line topology to the master. **A2** Change to CP2. **A3** Configure two connections (one connection for setting a value and one connection for receiving a value). **A4** Change to CP4 and activate connections. **A5** Connect a second slave from the testing base, which supports SCP_HP and can display the actual phase, at the end of the line. **A6** Hotplug the second slave.
Expected results	**E6** Slave should be in CP4.
Teardown actions	none
Workflow: Finished	

SCP Mux/ctm/00012

***Test* - 4_00012_Check SCP_Mux**

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00012
Link to specification	https://wiki.sercos-service.org/current/SCP_Mux
Full name	Check SCP_Mux
Description	Check if mutiplexing data to and from a slave is possible. To do this configure two connections (one for receiving and one for transmitting) with a device from the testing base and use these connections to multiplex data to an from the slave in CP4. The used container will be protocolled.
sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_Mux supported
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which supports SCP_Mux, to the master (and has an engineering port to display parameter during operation). **A2** Change to CP2. **A3** Configure two connections (one connection for multiplexing data to the slave and one connection for multiplexing data from the slave).

	A4 Select atleast two parameter from S-0-0445 IDN-list of configurable data in the MDT data container (if parameter is not supported use S-0-0188 IDN-list of configurable data as consumer instead) and atleast two parameter from S-0-0444 IDN-list of configurable data in the AT data container (if parameter is not supported use S-0-0187 IDN-list of configurable data as producer instead) and configure these parameter for multiplexing. **A4** Change to CP4 and activate connections. **A5** Multiplex first parameter to the slave. **A6** Multiplex second parameter to the slave. **A7** Display first multiplexed parameter from the slave. **A8** Display second multiplexed parameter from the slave. **A9** Protocol used container and parameter.
Expected results	**E5** The slave should react according to / display over engineering port the multiplexed data. **E6** The slave should react according to / display over engineering port the multiplexed data. **E7** First multiplexed parameter should be displayed correct (be equal to information shown over engineering port). **E8** Second multiplexed parameter should be displayed correct (be equal to information shown over engineering port).
Teardown actions	none
Workflow: Finished	

SCP RTB/ctm/00015

Test *- 5_00015_Check SCP_RTB*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00015
Link to	https://wiki.sercos-service.org/current/SCP_RTB

specification	
Full name	Check SCP_RTB
Description	Check if realtime data bits can be configured between slave and master. Configure two connections (one for receiving and one for transmitting data) with slave from testing base. In produced connection configure realtime bits that can be transmitted from the master to the slave and in the consumed connection configure realtime bits that can be transmitted from the slave to the master.
sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_RTB supported
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which supports SCP_RTB (and has an engineering port to display parameter during operation), to the master. **A2** Change to CP2. **A3** Configure two connections (one connection for setting a value and one connection for receiving a value). **A4** Configure randomly chosen parameter from S-0-0398 IDN list of configurable real-time/status bits and S-0-0399 IDN list of configurable real-time/control bits to according connection in S-0-1050.x.20 IDN Allocation of

	real-time bit. **A5** Set S-0-1050.x.21 Bit allocation of real-time bit in both connections to a reasonable value. **A6** Change to CP4 and activate connections. **A7** Documentate used parameter and bits.
Expected results	**E4** No error should be reported. **E5** No error should be reported. **E6** No error should be reported.
Teardown actions	none
Workflow: Finished	

SCP SMP/ctm/00017

Test *- 6_00017_Check SCP_SMP*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00017
Link to specification	https://wiki.sercos-service.org/current/SCP_SMP
Full name	Check SCP_SMP
Description	Check if master can send and receive messages over SMP correctly through configuring two connections (one consumed and one produced) which each include a SMP container. Use these container to transmit a message over SMP and then receive answer of the message. Then show that the answer of the SMP message has been received correctly.
sercos III revision	1.1.2

Automation level	manual
Test conditions	SCP_SMP supported
Attributes	
Preconditions	
Actions	**A1**
Expected results	**E1**
Teardown actions	none
Workflow: Declined	

SCP Sig/ctm/00016

Test *- 7_00016_Check SCP_Sig*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00016
Link to specification	https://wiki.sercos-service.org/current/SCP_Sig
Full name	Check SCP_Sig

Description	Check if it is possible to configure a signal control word and a signal status word correctly. For testing this two connections (one consumed and one produced) will be configured with a device from the testing base which include the signal status word and the signal control word in the corresponding connection. The signal control word is configured in a way so that a event can be started over the control word and the response then will be displayed in the status word.
sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_Sig supported
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which supports SCP_Sig (and has an engineering port to display parameter during operation), to the master. **A2** Change to CP2. **A3** Configure two connections (one connection with S-0-0144 Signal status word and one connection with S-0-0145 Signal control word). **A4** Configure randomly chosen parameter from S-0-0398 IDN list of configurable real-time/status bits and S-0-0399 IDN list of configurable real-time/control bits to S-0-0026 Configuration list for signal status word and S-0-0027 Configuration list for signal control word. **A5** Set S-0-0328 Bit number allocation list for signal status word and S-0-0329 Bit number allocation list for signal control word to a reasonable value.

	A6 Change to CP4 and activate connections. **A7** Documentate used parameter and bits.
Expected results	**E4** No error should be reported. **E5** No error should be reported. **E6** No error should be reported.
Teardown actions	none
Workflow: Finished	

SCP Sync/ctm/00010

Test *- 8_00010_Check SCP_Sync configuration*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00010
Link to specification	https://wiki.sercos-service.org/current/SCP_Sync
Full name	Check SCP_Sync configuration
Description	Check if master configures parameter for SCP_Sync correctly and realtime data can be exchanged. For testing this configure two connections (one consumed and one received) with a device from the testing base - that can display the loss of synchronisation - and change to CP4. In CP4 slave should still be synchronized (no synchronisation error should be displayed)
sercos III revision	1.1.2
Automation level	manual

Test conditions	SCP_Sync supported
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which supports SCP_Sync, can display the loss of synchronisation and has an engineering port to access parameter during operation, to the master. **A2** Change to CP2. **A3** Configure two connections (one connection for setting a value and one connection for receiving a value). **A4** Change to CP4 and activate connections. **A5** Wait a random time span.
Expected results	**E4** SCP_Sync parameter have been transfered by the master to the slave. **E5** Slave should not lose synchronisation.
Teardown actions	none
Workflow: Finished	

SCP VarCFG/ctm/00009

Test - 9_00009_Check SCP_VarCfg configuration

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00009
Link to specification	https://wiki.sercos-service.org/current/SCP_VarCFG

Full name	Check SCP_VarCfg configuration
Description	Configure two connections with a SCP_VarCfg slave (one connection for receiving data and one connection for transmitting data). Then check if realtime data exchange to and from the slave is working correct by setting and reading values in CP4.
sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_VarCfg supported
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base which supports SCP_VarCfg to the master. **A2** Change to CP2. **A3** Configure two connections (one connection for setting a value and one connection for receiving the feedback of the value set by the master). **A4** Change to CP4 and activate connections. **A5** Write a reasonable value to the slave over realtime data. **A6** Read feedback value of the slave over realtime data.
Expected results	**E5** Slave should have reacted accordingly (check through slave action or slave's engineering port).

	E6 Feedback value should match the state of the slave.
Teardown actions	none
Workflow: Finished	

SCP WD/ctm/00014

Test *- 10_00014_Check SCP_WD*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00014
Link to specification	https://wiki.sercos-service.org/current/SCP_WD
Full name	Check SCP_WD
Description	Check if a connection with a watchdog can be used by the master. For doing this configure two watchdog monitored connections (one consumed and one produced) with a slave that supports SCP_WD. Then change to CP4 and check if data can be transmitted and received.
sercos III revision	1.1.2
Automation level	manual
Test conditions	SCP_WD supported
Attributes	

Preconditions	
Actions	**A1** Connect a slave from the testing base, which supports SCP_WD (and has an engineering port to display parameter during operation), to the master. **A2** Change to CP2. **A3** Configure two connections (one connection for setting a value and one connection for receiving a value). **A4** Change to CP4 and activate connections. **A5** Send value to the slave. **A6** Depending on slave trigger event over engineering port for sending a value or wait for value when it is send by the application.
Expected results	**E5** Slave reacts on value / displays value over engineering port. **E6** Value should be plausible (cross check over engineering port).
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/DLPDU structure/AT DLPDU/AT Payload during initialization/CP0/ctm/00002

Test *- 11_00002_Check CP0 behaviour*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00002
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/DLPDU_structure/AT_DLPDU/AT_Payload_during_initialization/CP0
Full	Check CP0 behaviour

name	
Description	Check if master detects devices in CP0 correctly. For doing this connect as many slaves as supported by the master (or maximum available in testing laboratory) and check if all are recognized by the master. Do this depending on the configuration of the master in ring, single line and double line.
sercos III revision	1.1.2
Automation level	manual
Test conditions	none
Attributes	
Preconditions	
Actions	**A1** Foreach *topology* in [ring, single line, double line] do **A2** Connect *maximum number* of supported slaves from the testing base to the master in *topology* topology. **A3** Change to CP0. **A4** Check if *maximum number* of slaves have been detected. **A5** Change to NRT.

	A6 Documentate *maximum number* of slaves in *topology*. **A7** End foreach.
Expected results	**E4** *Maximum number* of slave have been detected.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/DLPDU structure/MDT DLPDU/MDT payload in normal operation/MDT real-time data field/ctm/00001

Test* - *12_00001_Realtime data exchange

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00001
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/DLPDU_structure/MDT_DLPDU/MDT_payload_in_normal_operation/MDT_real-time_data_field
Full name	Realtime data exchange
Description	Check if master can exchange realtime data with one device. For doing this connect a device from the testing base, with which the master can exchange realtime data and configure two connections (one consumed and one produced). Then bring device in CP4 and check if data can be send and received to and from the device. Test this depending on the abilitys of the master.
sercos III revisio	1.1.2

n	
Automation level	manual
Test conditions	none
Attributes	
Preconditions	
Actions	**A1** Connect a slave from testing base to the master, with that the master can exchange realtime data. **A2** Change to CP2. **A3** Configure one consumed and one produced connection (eg for setting outputs, moving drive and receiving feedback values). **A4** Documentate used configuration. **A5** Change to CP4 and activate connections. **A6** Visually check if realtime data is working correct.
Expected results	**E6** Slave is handling realtime data correct (eg. setting outputs, moving drive and feedback is displayed correct).
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/DLPDU structure/MDT DLPDU/MDT payload in normal operation/MDT real-time data field/ctm/00036

Test - 13_00036_realtime data stress test

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00036
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/DLPDU_structure/MDT_DLPDU/MDT_payload_in_normal_operation/MDT_real-time_data_field
Full name	realtime data stress test
Description	Configure a heavy amount of realtime data and then check if master still can exchange data. For doing this configure maximum amount of produced connections (supported by the master) with slaves from the testing base. Then change to CP4 and check if cyclic communication is possible without an error displayed in the slaves (e.g. new data not toggled).
sercos III revision	1.1.2
Automation level	manual
Test conditions	none

Attributes	
Preconditions	
Actions	**A1** Connect maximum number of supported slaves from the testing base to the master, with that the master can exchange realtime data and that have a LCD or a sercos LED to display errors. **A2** Change to CP2. **A3** Configure as many produced synchronous connections, with randomly chosen but reasonable connection cycle times, as possible between the master and the slaves. **A4** Change to CP4 and activate connections. **A5** Wait some time. **A6** Visually check if slaves display an error.
Expected results	**E6** Master is toggling *New data* correct and no error occurs.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/DL management/Enable and disable cyclic communication/Communication phases (CP)/Communication phase 0 (CP0)/ctm/00003

Test* - *14_00003_Non projection of existing device

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00003

Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/DL_management/Enable_and_disable_cyclic_communication/Communication_phases_(CP)/Communication_phase_0_(CP0)
Full name	Non projection of existing device
Description	Check if an error is produced during phase change if a slave is not projected that does exist. For doing this project slaves (3-4 from testing base) that are connected to the master and in addition one slave (with engineering port) which is connected to the master but not projected. Then change to CP4 and check if an error is reported or if the non projected device has been configured automatically.
sercos III revision	1.1.2
Automation level	manual
Test conditions	none
Attributes	
Preconditions	

Actions	**A1** Connect 5 slaves from testing base (at least one with an engineering port for parameter access) to the master. **A2** Project 4 slaves in the master, but not the slave with the engineering port. **A3** Change to CP4. **A4** Check over engineering port if slave has been configured automatically.
Expected results	**E3** An error should be reported before CP4 -> PASSED. **E4** Slave has been configured automatically -> PASSED.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/DL management/Enable and disable cyclic communication/Communication phases (CP)/Communication phase 2 (CP2)/ctm/4/00004

Test* - *15_00004_Failed CP3 transition check

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00004
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/DL_management/Enable_and_disable_cyclic_communication/Communication_phases_(CP)/Communication_phase_2_(CP2)
Full name	Failed CP3 transition check
Descr	Check if master detects failed procedure command "CP3 transition check" and stays in

iption	CP2. For doing this one slave is connected to the master and a configuration is written to the slave that will result in a S-0-0127 CP3 transition check error. Then try to change to CP3. After error has occured check if master has stayed in CP2.
sercos III revision	1.1.2
Automation level	manual
Test conditions	none
Attributes	
Preconditions	
Actions	**A1** Connect a slave (which has a engineering port and software for parameter access during operation) from testing base to the master. **A2** Change to CP2. **A3** Over engineering port change configuration to fail during S-0-0127 CP3 transition check (manufacturer specific). **A4** Documentate used configuration. **A5** Try to change to CP3.
Expected	**E2** Slave is in CP2.

results	**E5** Master stays in CP2 and may display an error (cross check if slave displays an error).
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/DL management/Enable and disable cyclic communication/Communication phases (CP)/Communication phase 3 (CP3)/ctm/00005

Test - 16_00005_Failed CP4 transition check

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00005
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/DL_management/Enable_and_disable_cyclic_communication/Communication_phases_(CP)/Communication_phase_3_(CP3)
Full name	Failed CP4 transition check
Description	Check if master deletes failed procedure command "CP4 transition check" and stays in CP3. For doing this one slave is connected to the master and a configuration is written to the slave that will result in a S-0-0128 CP4 transition check error. Then try to change to CP4. After error has occured it will be checked if master stayed in CP3.
sercos III revision	1.1.2

Automation level	manual
Test conditions	none
Attributes	
Preconditions	
Actions	**A1** Connect a slave (which has a engineering port and software for parameter access during operation) from testing base to the master. **A2** Change to CP2 and configure a connection so that the slave would be able to change to CP4. **A3** Over engineering port change slave configuration to fail during S-0-0128 CP4 transition check (manufacturer specific). **A4** Documentate used configuration. **A5** Try to change to CP4.
Expected results	**E2** Slave is in CP2 and connection are configured. **E5** Master stays in CP3 and may display an error (cross check if slave displays an error).
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/Data transmission methods/SVC/ctm/00037

Test* - *17_00037_Data exchange with maximum of supported slaves over SVC

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00037
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/Data_transmission_methods/SVC
Full name	Data exchange with maximum of supported slaves over SVC
Description	Check if master can exchange data with maximum number of supported slaves. For doing this connect the maximum number of slaves that are supported by the master (or are available in the testing laboratory) to the master. Then bring ring in CP2 and read known values out of randomly chosen slaves over SVC and check if the values are correct. Check the correct behaviour of SVC also in CP4.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	

Preconditions	
Actions	**A1** Connect maximum number of supported slaves from the testing base to the master. **A2** Change to CP2. **A3** Foreach 1..4 randomly chosen slaves do **A4** Read value of S-0-1300.x.05 Vendor Device ID. **A5** End foreach. **A6** Change to CP4. **A7** Foreach 1..4 randomly chosen slaves do **A8** Read value of S-0-1300.x.05 Vendor Device ID. **A9** End foreach.
Expected results	**E4** Read value fits to device. **E8** Read value fits to device.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00018

Test - 18_00018_Open non existing IDN

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00018
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification

	/IDN_-_Identification_numbers/IDN_specification
Full name	Open non existing IDN
Description	Try to open an IDN that does not exist in the slave and check if master reports an error. For doing this connect a slave from the testing base to the master and bring sercos communication in CP2. Then try to read IDN that does not exist and check if an error is displayed.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base to the master. **A2** Change to CP2. **A3** Try to read an IDN that does not exist in slave.
Expected results	**E3** Master should display an error.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00020

Test - 19_00020_Check write protection

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00020
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check write protection
Description	Try to write parameter that are write protected in CP2, CP3 and CP4 and check if correct error message is displayed.
sercos III revision	1.1.2
Automation level	manual
Test conditions	none
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base to the master.

	A2 Foreach *phase* in *master_accessable_phases* do **A3** Change to *phase*. **A4** Try to write a random value to S-0-0021 IDN-list of invalid operation data for CP2. **A5** End foreach.
Expected results	**E4** Error message is displayed in the master.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00021

Test* - *20_00021_Check decimal point

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00021
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check decimal point
Description	Check if the decimal point is displayed and if the position of the decimal point is correct. For doing this read a known parameter with decimal point and check if the displayed value is correct.
sercos III revision	1.1.2

Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation, to the master. **A2** Change to CP2. **A3** Change S-0-1002 Communication Cycle time (tScyc) over engineering port to 1000,000 µs. **A4** Read S-0-1002 Communication Cycle time (tScyc) over master. **A5** Change S-0-1002 Communication Cycle time (tScyc) over engineering port to 62,500 µs. **A6** Read S-0-1002 Communication Cycle time (tScyc) over master.
Expected results	**E4** Read value is equal 1000,000 µs. **E6** Read value is equal 62,500 µs.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00022

Test - 21_00022_Check data type and display format

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00022
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check data type and display format
Description	Read parameter with different data types and display formats and check that all formats are provided correctly. For testing this different parameters are selected. The master needs to provide the parameter how they are exspected.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base to the master. **A2** Change to CP2. **A3** Read S-0-0127 CP3 transition check.

	A4 Read S-0-1002 Communication Cycle time (tScyc). **A5** Read S-0-0036 Velocity command value. **A6** Read S-0-1009 Device Control (C-Dev) offset in MDT. **A7** Read S-0-1300.x.05 Vendor Device ID. **A8** Read S-0-1050.x.06 Configuration List. **A9** Read vendor specific float IDN (no official float IDN exists for this test) **A10** Read S-0-1305.0.01 sercos current time.
Expected results	**E3** Read value could be interpreted as binary. **E4** Read value could be interpreted as unsigned decimal. **E5** Read value could be interpreted as signed decimal. **E6** Read value could be interpreted as hexadecimal. **E7** Read value could be interpreted as text (UTF8 the null termination (\0) shall not be used). **E8** Read value could be interpreted as IDN. **E9** Read value could be interpreted as float. **E10** Read value could be interpreted according to IEC 61588 4 octets seconds & 4 octets nano seconds, starts with 1.1.1970 computed in UTC.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00023

Test - 22_00023_Check procedure command execution

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00023

Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check procedure command execution
Description	Execute a procedure command and check if result of procedure command is displayed correctly. For testing this the master needs to bring one randomly chosen slave in CP2 and then execute S-0-0127. Depending on the configuration the reaction should be displayed.
sercos III revision	1.1.2
Automation level	manual
Test conditions	none
Attributes	
Preconditions	
Actions	**A1**
Expected results	**E1**
Teardown actions	none

Workflow: Declined

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00024

Test *- 23_00024_Check data length*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00024
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check data length
Description	Check if data length is displayed correctly and that only parameter can be insert that are within the data length. Check this for all possible data length. For testing this bring one randomly chosen slave into CP2 and first check if data length is displayed correctly. Then try to write a too big value to the parameter. This will be done for all data types.
sercos III revision	1.1.2
Automation level	manual
Test conditions	none
Attributes	

Preconditions	
Actions	**A1** Connect a slave from the testing base to the master. **A2** Change to CP2. **A3** Read S-0-1009 Device Control (C-Dev) offset in MDT. **A4** Write **0x1 0000** to S-0-1009 Device Control (C-Dev) offset in MDT. **A5** Read S-0-1003 Allowed MST losses in CP3/CP4. **A6** Write **4294967296** to S-0-1003 Allowed MST losses in CP3/CP4. **A7** Read S-0-1305.0.01 sercos current time. **A8** Write **0x1 0000 0000 0000 0000** to S-0-1305.0.01 sercos current time. **A9** Read S-0-0142 Application type. **A10** Write **ä** to S-0-0142 Application type. **A11** Read S-0-1010 Lengths of MDTs. **A12** Write **65536** to S-0-1010 Lengths of MDTs. **A13** Write **S-0-1002** to S-0-1050.x.06 Configuration List. **A14** Read S-0-1050.x.06 Configuration List. **A15** Write **V-0-1002** to S-0-1050.x.06 Configuration List. **A16** Write **S-8-1002** to S-0-1050.x.06 Configuration List. **A17** Write **S-0-10020** to S-0-1050.x.06 Configuration List. **A18** Write **S-0-1002.256.0** to S-0-1050.x.06 Configuration List. **A19** Write **S-0-1002.0.256** to S-0-1050.x.06 Configuration List.
Expected results	**E3** Value should be displayed as 2 byte value.

	E4 Write error should be displayed. **E5** Value should be displayed as 4 byte value. **E6** Write error should be displayed. **E7** Value should be displayed as 8 byte value. **E8** Write error should be displayed. **E9** Value should be displayed as 1 byte variable value. **E10** Written value should be aceepted and displayed. **E11** Value should be displayed as 2 byte variable value. **E12** Write error should be displayed. **E14** Value should be displayed as IDN. **E15** Write error should be displayed. **E16** Write error should be displayed. **E17** Write error should be displayed. **E18** Write error should be displayed. **E19** Write error should be displayed.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00025

Test - 24_00025_Check conversion factor

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00025

Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check conversion factor
Description	Check if the conversion factor is used correctly (convert numeric data to display format). For checking this bring a randomly chosen slave in CP2 and then read a parameter which uses a conversion factor. Then check if parameter is displayed correctly.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation and conversion factor unequal to one in some parameters, to the master. **A2** Change to CP2. **A3** Read values of a random parameterX (with conversion factor unequal 1) over engineering port. **A4** Read values of same parameterX over master.

Expected results	**E4** Value read is equal value read in **A3** .
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00026

Test *- 25_00026_Check unit*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00026
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check unit
Description	Check if master displays unit of parameter correct. To test this bring a randomly chosen slave from the testing base in CP2 and read a parameter which has a display unit. Compare the displayed unit with the unit that was exspected to be found.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters

Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation, to the master. **A2** Change to CP2. **A3** Foreach 1..4 randomly chosen parameter with unit do **A4** Read unit of parameterX over engineering port. **A5** Read unit of parameterX over master. **A6** Documentate used parameterX. **A7** End foreach.
Expected results	**E5** Read unit is equal read unit in **A4**.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00027

Test* - *26_00027_check min/max values

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00027
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification

Full name	check min/max values
Description	Check if master displays min and max values correct. For testing this bring a randomly chosen slave from the testing base in CP2 and read a parameter which has min and max values. Then compare the displayed values of the master with the values that were expected to be found.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation and min/max values in some parameters, to the master. **A2** Change to CP2. **A3** Read min/max values of a random parameterX (with min/max values) over engineering port. **A4** Read min/max values of same parameterX over master.
Expected results	**E4** Min/max value is equal to min/max value read in **A3**.
Teardown actions	none

Workflow: Finished

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00028

Test* - *27_00028_Check reading operation data

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00028
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check reading operation data
Description	Read parameters with fixed length 2/4/8 octets and variable length with list elements of 1, 2, 4 and 8 octets up to 65 532 octets and check if all operation data is displayed correct. For testing a randomly chosen slave from testing base will be used that is in CP2.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	
Precondition	

s	
Actions	**Comment: If a parameter does not exist in the slave chose a different parameter with same length and documentate which parameter has been used.** **A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation, to the master. **A2** Change to CP2. **A3** Read S-0-1009 Device Control (C-Dev) offset in MDT over engineering port. **A4** Read S-0-1009 Device Control (C-Dev) offset in MDT over master. **A5** Read S-0-1002 Communication Cycle time (tScyc) over engineering port. **A6** Read S-0-1002 Communication Cycle time (tScyc) over master. **A7** Read S-0-1305.0.01 sercos current time over engineering port. **A8** Read S-0-1305.0.01 sercos current time over master. **A9** Read S-0-1300.x.05 Vendor Device ID over engineering port. **A10** Read S-0-1300.x.05 Vendor Device ID over master. **A11** Read S-0-1010 Lengths of MDTs over engineering port. **A12** Read S-0-1010 Lengths of MDTs over master. **A13** Read S-0-1050.x.06 Configuration List over engineering port. **A14** Read S-0-1050.x.06 Configuration List over master. **A15** Read S-0-1500.x.03 List of module type codes over engineering port. **A16** Read S-0-1500.x.03 List of module type codes over master.
Expected results	**E4** Read value is equal to read value in **A3**. **E6** Read value is equal to read value in **A5**. **E8** Read value is equal to read value in **A7**.

	E10 Read value is equal to read value in **A9**. **E12** Read value is equal to read value in **A11**. **E14** Read value is equal to read value in **A13**. **E16** Read value is equal to read value in **A15**.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00029

***Test* - 28_00029_Check writing operation data**

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00029
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check writing operation data
Description	Write parameters with fixed length 2/4/8 octets and variable length with list elements of 1, 2, 4 and 8 octets up to 65 532 octets and check if all operation data is set in the slave. Test this with a randomly chosen slave from the testing base which is in CP2.
sercos III revision	1.1.2
Automation level	manual

Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**Comment: If a parameter does not exist in the slave chose a different parameter with same length and documentate which parameter has been used.** **A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation, to the master. **A2** Change to CP2. **A3** Write plausibel value to S-0-1009 Device Control (C-Dev) offset in MDT. **A4** Read S-0-1009 Device Control (C-Dev) offset in MDT over engineering port. **A5** Write plausibel value to S-0-1002 Communication Cycle time (tScyc). **A6** Read S-0-1002 Communication Cycle time (tScyc) over engineering port. **A7** Write plausibel value to S-0-1305.0.01 sercos current time. **A8** Read S-0-1305.0.01 sercos current time over engineering port. **A9** Write plausibel value to S-0-0142 Application Type. **A10** Read S-0-0142 Application Type over engineering port. **A11** Write plausibel value to S-0-1010 Lengths of MDTs. **A12** Read S-0-1010 Lengths of MDTs over engineering port. **A13** Write plausibel value to S-0-1050.x.06 Configuration List. **A14** Read S-0-1050.x.06 Configuration List over engineering port.
Expected	**E4** Read value is equal to written value in **A3**.

results	**E6** Read value is equal to written value in **A5**. **E8** Read value is equal to written value in **A7**. **E10** Read value is equal to written value in **A9**. **E12** Read value is equal to written value in **A11**. **E14** Read value is equal to written value in **A13**.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00030

Test *- 29_00030_Check correct order of list*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00030
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check correct order of list
Description	Read list and check if correct order in list is preserved. For testing this read parameter S-0-0017 IDN-list of all operation data and check if sercos order for displaying parameter lists is preserved.
sercos III revision	1.1.2
Automation	manual

level	
Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base to the master. **A2** Change to CP2. **A3** Read parameter S-0-0017 IDN-list of all operation data.
Expected results	**E3** Result should be sorted by IDN.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00031

Test *- 30_00031_Writing incorrect data*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00031
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Writing incorrect data

Description	Write wrong type of data to a slave and check master behaviour. For testing this bring a randomly chosen slave in CP2 and try to write text to a parameter which exspects a numeric value. The GUI or the master itself needs to react with an error.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation, to the master. **A2** Change to CP2. **A3** Write the value 1 ms to S-0-1002 Communication Cycle time (tScyc). **A4** Read value of S-0-1002 Communication Cycle time (tScyc) over engineering port. **A5** Write the value "**FP**" to S-0-1002 Communication Cycle time (tScyc). **A6** Read value of S-0-1002 Communication Cycle time (tScyc) over engineering port.
Expected results	**E4** Read value is equal 1 ms. **E5** Master should display an error message.

	E6 Read value should be 1 ms.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/00032

***Test* - 31_00032_Check IDN name**

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00032
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Check IDN name
Description	Check if the IDN name is displayed correctly. For checking this bring a randomly chosen slave from the testing base in CP2 and then read an IDN. Then check if IDN name is displayed correctly.
sercos III revision	1.1.2
Automation level	manual
Test conditions	master supports free access of slave parameters
Attributes	

Preconditions	
Actions	**A1** Connect a slave from the testing base, which has a engineering port for parameter access during operation, to the master. **A2** Change to CP2. **A3** Foreach 1..4 randomly chosen parameter with name do **A4** Read name of parameterX over engineering port. **A5** Read name of parameterX over master. **A6** End foreach.
Expected results	**E5** Read name is equal read name in **A4**.
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/IDN - Identification numbers/IDN specification/ctm/19/00019

Test *- 32_00019_Open IDN*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00019
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/IDN_-_Identification_numbers/IDN_specification
Full name	Open IDN
Description	Read IDN with and without structure instance and structure element. For doing this connect a randomly chosen slave to the master and bring sercos

	communication in CP2. Then open an IDN without structure instance and structure element (sercos II parameter) and afterwards open an IDN with structure instance and structure element (sercos III). Check if both parameter have been displayed correct.
sercos III revision	1.1.2
Automation level	manual
Test conditions	none
Attributes	
Preconditions	
Actions	**A1** Connect a slave from the testing base, which has an engineering port for parameter access during operation, to the master. **A2** Change to CP2. **A3** Try to read IDN S-0-0017 IDN-list of all operation data from slave. **A4** Try to read IDN S-0-0017 IDN-list of all operation data through engineering port. **A5** Try to read IDN S-0-1050.x.05 Current length of connection from the slave. **A6** Try to read IDN S-0-1050.x.05 Current length of connection through engineering port.
Expected results	**E4** S-0-0017 IDN-list of all operation data should be equal value read in **A3**. **E6** S-0-1050.x.05 Current length of connection should be equal value read in **A5**.

Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/Telegram timing and DLPDU handling/Communication mechanisms/Redundancy of RT-communication with ring topology/ctm/00007

Test *- 33_00007_Check ring break and recovery*

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00007
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/Telegram_timing_and_DLPDU_handling/Communication_mechanisms/Redundancy_of_RT-communication_with_ring_topology
Full name	Check ring break and recovery
Description	Check if a ring break is detected by the master and the correct conclusion is executed. Further check if ring recovery is working correctly. For testing this connect two slaves from the testing base to the master and bring the sercos communication in CP1. Then disconnect the wiring between the slaves and check if an error is displayed. Then reconnect the wires and try to heal the ring and check if both slaves are in Fast-Forward again. Check this for all CPs where this functionality is supported by the master.
sercos III revision	1.1.2
Automation	manual

level	
Test conditions	Ring break and recovery supported
Attributes	
Preconditions	
Actions	**A1** Connect two slaves (which have a engineering port for parameter access during operation) from testing base in ring topology to the master. **A2** Foreach *master_accessable_phases* do **A3** Change to *master_accessable_phases*. **A4** Removing wires between slaves. **A5** Reconnect wires between slaves. **A6** Heal sercos ring. **A7** End foreach.
Expected results	**E4** Error is displayed in the master. **E6** Both sercos slaves should be in Fast-Forward again (check through S-0-1045 Device Status or Wireshark).
Teardown actions	none
Workflow: Finished	

Spec3/Data link layer – Protocol specification/Telegram timing and DLPDU handling/Usage of real-time channel with different network topologies/ctm/00006

Test - 34_00006_Adding and removing slave during operation

Link to test case	https://wiki.sercos-service.org/current/Jumpto/ctm/00006
Link to specification	https://wiki.sercos-service.org/current/Spec3/Data_link_layer_%E2%80%93_Protocol_specification/Telegram_timing_and_DLPDU_handling/Communication_mechanisms/Redundancy_of_RT-communication_with_ring_topology
Full name	Adding and removing slave during operation
Description	Check the behaviour of the master if a slave at the end of the line is removed during operation and plugged in again afterwards. For testing this bring a line of two randomly chosen slaves - without hotplug - in CP2. Then remove the last slave in line and check if master reports an error. Then add the slave again and check that the slave is not part of the communication without hotplugging it.
sercos III revision	1.1.2
Automation level	manual
Test conditions	Master supports removing slaves during operation
Attributes	

Preconditions	
Actions	**A1** Connect two slaves from testing base, which dont support hotplug, in *Single Line Topology* to the master. **A2** Change to CP2. **A3** Remove slave at end of line. **A4** Add slave at end of line again. **A5** Try to access just added slave over SVC.
Expected results	**E3** Error is displayed in the master. **E5** Master is not able to access just added slave.
Teardown actions	none
Workflow: Finished	